UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA
Departamento de Sistemas Informáticos y Computación

ALGORITMOS HEURÍSTICOS Y APLICACIONES A MÉTODOS FORMALES.

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Pablo M. Rabanal Basalo

Bajo la dirección de los doctores

Ismael Rodríguez Laguna
Fernando Rubio Diez

Madrid, 2010

ISBN: 978-84-693-9498-4

Algoritmos heurísticos y aplicaciones a métodos formales

TESIS DOCTORAL

FEBRERO 2010

Autor: Pablo M. Rabanal Basalo

Departamento de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

Directores: Ismael Rodríguez Laguna y Fernando Rubio Diez

Resumen

Los algoritmos de optimización basados en búsquedas locales recorren el espacio de soluciones tratando de conseguir una buena solución en un tiempo razonable para minimizar o maximizar un valor y tratando de evitar *quedarse estancado* en mínimos o máximos locales. Para ello parten de una solución y la modifican aplicando ciertos operadores para calcular soluciones *vecinas* que mejoren la calidad de la solución inicial. Normalmente, estas técnicas de búsqueda se aplican a problemas NP-duros en los que el espacio de búsqueda es muy grande y es necesario el uso de funciones *heurísticas* para eliminar rutas de búsqueda no prometedoras. El problema de estas heurísticas es que también se pueden desechar rutas que lleven a buenas soluciones. Entre estos métodos cabe destacar la escalada, el enfriamiento simulado, los algoritmos genéticos, los algoritmos de optimización basados en nubes de partículas o los algoritmos de colonias de hormigas.

Un tema al que se han aplicado métodos evolutivos de manera exitosa en los últimos años son los métodos formales. Los métodos formales son técnicas que típicamente han sido aplicadas tanto a la especificación formal como a la verificación formal de sistemas software, con la idea de desarrollar especificaciones claras, concisas y ausentes de ambigüedades. El punto de encuentro entre dos áreas tan diferentes es debido a que los métodos formales se encuentran comúnmente con un problema en la práctica: deben analizar sistemas en los que el número de estados de la especificación crece exponencialmente. Es aquí donde las técnicas heurísticas proporcionan estrategias eficientes que se pueden aplicar para intentar buscar errores potenciales en el sistema.

En esta tesis se introduce una nueva técnica evolutiva llamada *River Formation Dynamics* inspirada en la naturaleza y basada en el proceso geológico de la formación de los ríos. Primero se diseña un algoritmo básico basado en estas ideas para posteriormente aplicarlo a resolver problemas NP-completos de diferente índole. Uno de los problemas a los que se ha aplicado este método para probar su funcionamiento es el problema del *viajante de comercio*. Además se han definido nuevos problemas NP-completos como son los casos del problema del *árbol recubridor mínimo* y el *árbol de distancias mínimas* en grafos de costes variables. Para resolver estos problemas es necesario adaptar el algoritmo básico a cada caso. También se ha aplicado River Formation Dynamics a escenarios típicos de métodos formales donde se ha utilizado esta técnica para alcanzar ciertos estados/transiciones de una especificación definida por una máquina de estados finitos.

Agradecimientos

En esta sección aprovecharé para darles las gracias a todos aquellos que me ayudaron desinteresadamente. Comenzaré por agradecer el tiempo, esfuerzo y dedicación a mis directores de tesis: Fernando e Ismael, ya que el desarrollo de esta tesis nunca hubiese sido posible sin su ayuda y dirección. También agradecer el apoyo a los compañeros del grupo de investigación al que pertenezco: Grupo de Testing y Evaluación del Rendimiento.

En estos momentos también quiero acordarme de mi novia, Vicky, *simplemente* por estar ahí y soportar mis fines de semana de deadline; de mi madre, que desde el primer día que empecé a escribir la tesis me preguntaba: ¿has acabado ya la tesis?, ¿cuándo acabas la tesis?, estás tardando mucho, ¿no?; y, por supuesto, de mi padre.

Acabar la sección con tres citas célebres de los hermanos Marx:

Es mejor permanecer callado y parecer tonto que hablar y despejar las dudas definitivamente.

¿A quién va usted a creer?, ¿a mí, o a sus propios ojos?

Claro que lo entiendo. Incluso un niño de cinco años podría entenderlo. ¡Qué me traigan un niño de cinco años!

Lista de Publicaciones Originales

Esta tesis doctoral se ha presentado según el formato de publicaciones y ha dado lugar a un conjunto de publicaciones originales enumeradas a continuación.

1. P. Rabanal, I. Rodríguez, and F. Rubio. Using river formation dynamics to design heuristic algorithms. In *UC'07: 6th international conference on Unconventional Computation, LNCS 4618*, pages 163–177. Springer, 2007.
   http://www.springerlink.com/content/w2453h31g9527157/

2. P. Rabanal, I. Rodríguez, and F. Rubio. Finding minimum spanning/distances trees by using River Formation Dynamics. In *ANTS'08: 6th international conference on Ant Colony Optimization and Swarm Intelligence, LNCS 5217*, pages 60–71. Springer, 2008.
   http://www.springerlink.com/content/hv760378624684u3/

3. P. Rabanal, I. Rodríguez, and F. Rubio. Solving dynamic TSP by using river formation dynamics. In *ICNC'08: 4th International Conference on Natural Computation*, pages 246–250. IEEE Computer Society, 2008.
   http://www.computer.org/portal/web/csdl/doi/10.1109/ICNC.2008.760

4. P. Rabanal, I. Rodríguez, and F. Rubio. Applying river formation dynamics to solve NP-complete problems. In R. Chiong, editor, *Nature-Inspired Algorithms for Optimisation*, volume 193 of *Studies in Computational Intelligence*, pages 333–368. Springer, 2009.
   http://www.springerlink.com/content/q2g3v408666q1773/

5. P. Rabanal and I. Rodríguez. Hybridizing river formation dynamics and ant colony optimization. In *ECAL'09: 10th European Conference on Artificial Life*. Springer, in press.

6. P. Rabanal and I. Rodríguez. Testing restorable systems by using RFD. In *IWANN'09: 10th International Work-Conference on Artificial Neural Networks, LNCS 5517*, pages 351–358. Springer, 2009.
   http://www.springerlink.com/content/q2j2j673182mjlw3/

7. P. Rabanal, I. Rodríguez, and F. Rubio. A formal approach to heuristically test restorable systems. In *ICTAC'09: 6th International Colloquium on Theoretical Aspects of*

Computing, LNCS 5684, pages 292–306. Springer, 2009.
http://www.springerlink.com/content/qn024854771052w4/

8. P. Rabanal, I. Rodríguez, and F. Rubio. Applying RFD to construct optimal quality-investment trees. Technical report, 2009.
http://kimba.mat.ucm.es/prabanal/research/jucs09.pdf, actualmente en proceso de revisión en la revista Journal of Universal Computer Science.

9. P. Rabanal, I. Rodríguez, and F. Rubio. Testing restorable systems: Formal definition and heuristic solution based on river formation dynamics. Technical report, 2010.
http://kimba.mat.ucm.es/prabanal/research/stvr10.pdf, actualmente en proceso de revisión en la revista Software Testing, Verification and Reliability.

Índice general

Capítulo 1

Introducción

El principal objetivo de esta tesis es la aplicación de métodos heurísticos inspirados en la naturaleza para resolver problemas de optimización NP-completos en general y, en particular, algunos relacionados con los *métodos formales de testing*. Más precisamente, utilizaremos métodos de optimización basados en colonias de hormigas y presentaremos una nueva heurística basada en la formación dinámica de los ríos. Aunque los algoritmos clásicos han conseguido resolver multitud de problemas eficientemente, hay una serie de problemas complejos, como es el caso de los problemas NP-completos, que los métodos convencionales no consiguen solucionar en la práctica. Dado que se cree que los problemas NP-completos necesitan tiempo exponencial en el caso peor para poder calcular una solución óptima, optamos por utilizar algoritmos heurísticos que permitan calcular una solución subóptima lo suficientemente buena en tiempo polinómico. Para comprobar la calidad de las soluciones del nuevo método heurístico basado en la formación de los ríos, lo aplicamos a diferentes problemas y lo comparamos con las soluciones obtenidas por los métodos basados en colonias de hormigas. Comenzamos resolviendo un problema clásico como *el Problema del Viajante de Comercio*, tanto en su versión estándar como en su versión dinámica (en la que pueden aparecer/desaparecer nodos y/o aristas a lo largo de la ejecución). Más tarde buscamos soluciones a problemas útiles en el entorno de un método formal como es el *testing de software*. Así definimos el problema del *Árbol de Distancias Mínimo* y el problema del *Árbol Recubridor Mínimo* para un grafo de coste variable, en el que el coste de una arista depende del camino recorrido antes de tomar esta arista. Así mismo definimos el problema de la *Secuencia de Carga Mínima* donde dada una especificación definida por una máquina de estados finitos queremos encontrar un plan que interactúe con el sistema y permita alcanzar un conjunto de estados y/o transiciones en el mínimo tiempo posible donde se permite guardar/restaurar configuraciones previas,

asumiendo un coste predeterminado.

Esta tesis se estructura de la siguiente manera. En el capítulo 2, realizaremos una introducción a los principales esquemas de optimización basados en búsquedas locales como la *escalada*, el *enfriamiento simulado*, los *algoritmos genéticos*, la *optimización basada en nubes de partículas* o los *algoritmos de colonias de hormigas*. Dicha introducción servirá para familiarizarnos con las principales características de estos métodos y comprobar el estado del arte. Estos algoritmos tratan de mejorar una solución inicial aplicando distintos operadores o funciones. Es por ello que para cada solución encontrada es necesario estimar su calidad utilizando una función. Estos métodos se aplican en los casos en los que el espacio de soluciones a considerar es muy grande y no se puede realizar un estudio sistemático de todas las soluciones porque conllevaría un tiempo impracticable. En particular utilizan unas funciones para limitar la búsqueda a ciertas regiones en las que se espera obtener la mejor solución posible. Para cerrar el capítulo describimos de forma detallada nuestro método heurístico basado en la formación de los ríos.

Tras este estudio, en el capítulo 3 analizaremos aplicaciones de los métodos evolutivos presentados a los *Métodos Formales*, más específicamente al *Testing Formal* y al *Model Checking*. Los métodos formales proveen métodos sistemáticos para analizar la corrección de los sistemas. Aunque los métodos formales se han aplicado de manera satisfactoria a multitud de problemas industriales, estos métodos típicamente se encuentran con un problema en la práctica, pues el número de estados a analizar de modo sistemático crece de forma exponencial con el tamaño del sistema a analizar. Es por ello que las técnicas exhaustivas para encontrar errores en los sistemas se sustituyen por estrategias heurísticas que permitan focalizar la búsqueda de errores potenciales en alguna característica crítica o sospechosa. De este modo, ilustraremos con numerosos ejemplos significativos cómo los métodos heurísticos y los métodos formales pueden trabajar de forma conjunta en este medio. En particular, los métodos evolutivos proveen estrategias eficientes para buscar buenas soluciones en el tipo de problemas que aparecen en los métodos formales como son el *Testing Formal* y el *Model Checking*.

En el capítulo 4 esquematizaremos y mostraremos las ideas generales de los artículos presentados en la tesis. Finalmente, en el capítulo 5 se presentan las conclusiones y el trabajo futuro que queda por delante, mientras que en el apéndice A se describen todos los algoritmos utilizados en los experimentos y en el apéndice B se adjuntan los artículos.

A continuación exponemos los índices de calidad de los artículos que conforman la tesis:

1. Using River Formation Dynamics to Design Heuristic Algorithms [80] fue publicado en *Unconventional Computation* (UC) en el año 2007 por Springer. Este congreso es el *sucesor* del desaparecido *Unconventional Models of Computation* que aparece en el Conference Ranking evaluado con 0,82 (puesto 101 de 788 en la categoría Applications / Education / Software / Theory / Communications / Graphics / Bioinformatics). En el índice CORE aparece evaluado como C.

2. Finding Minimum Spanning/Distances Trees by using River Formation Dynamics [81] fue publicado en *Ant Colony Optimization and Swarm Intelligence* (ANTS) en el año 2008 por Springer. Este congreso aparece en el Conference Ranking evaluado con 0,51 (puesto 66 de 701 en la categoría Artificial Intelligence / Machine Learning / Robotics / Human Computer Interaction) y en el índice CORE evaluado como B.

3. Solving Dynamic TSP by Using River Formation Dynamics [86] fue publicado en *International Conference on Natural Computation* (ICNC) en el año 2008 por IEEE Computer Society. Este congreso aparece en el índice CORE evaluado como C.

4. Applying River Formation Dynamics to Solve NP-Complete Problems [83] fue publicado como un capítulo del libro *Nature-Inspired Algorithms for Optimisation* en el año 2009 por Springer.

5. Hybridizing River Formation Dynamics and Ant Colony Optimization [85] fue aceptado en el *European Conference on Artificial Life* (ECAL) en el año 2009. Springer, in press. Este congreso aparece en el Conference Ranking evaluado con 0,53 (puesto 64 de 701 en la categoría Artificial Intelligence / Machine Learning / Robotics / Human Computer Interaction) y en el índice CORE evaluado como B.

6. Testing Restorable Systems by Using RFD [79] fue publicado en *International Work-Conference on Artificial Neural Networks* (IWANN) en el año 2009 por Springer. Este congreso aparece en el Conference Ranking evaluado con 0,55 (puesto 55 de 701 en la categoría Artificial Intelligence / Machine Learning / Robotics / Human Computer Interaction) y en el índice CORE evaluado como B.

7. A Formal Approach to Heuristically Test Restorable Systems [87] fue publicado en *International Colloquium on Theoretical Aspects of Computing* (ICTAC) en el año 2009 por Springer. Este congreso aparece en el índice CORE evaluado como B.

8. Applying RFD to Construct Optimal Quality-Investment Trees [82] se encuentra actualmente en proceso de revisión en la revista *Journal of Universal Computer Science*.

9. Testing Restorable Systems: Formal Definition and Heuristic Solution based on River Formation Dynamics [84] se encuentra actualmente en proceso de revisión en la revista *Software Testing, Verification and Reliability*.

Capítulo 2

Algoritmos de Optimización Basados en Búsquedas Locales

En estas primeras páginas se realizará un estudio de distintos *métodos de optimización basados en búsquedas locales*. Estas búsquedas recorren el espacio de soluciones e intentan alcanzar la mejor solución posible en un tiempo razonable, ya sea para minimizar la solución, ya sea para maximizarla. Para ello, una función evalúa la calidad de la solución actual en cada momento [14].

Estos métodos toman como punto de partida una solución inicial dada que intentarán mejorar aplicando ciertos operadores o funciones definidos por el usuario. Estos operadores se aplicarán a la solución actual para calcular otras soluciones similares (vecinas) por las que continuar la búsqueda. Se supone pues que, modificando una solución del problema, se pueden calcular soluciones mejores a una encontrada anteriormente, es decir, se pretende mejorar dicha solución paso a paso.

Este tipo de métodos se aplican cuando el tamaño del espacio de soluciones es muy grande y no se puede realizar una exploración sistemática que lo recorra completamente para hallar la mejor solución, pues se tardaría un tiempo inaceptable. Para reducir el espacio de búsqueda se utilizarán funciones heurísticas con el fin de *podarlo*, es decir, que ayudarán a eliminar caminos que lleven a soluciones de peor calidad que la solución actual. Dichas funciones tienen los inconvenientes de que se pueden *podar* caminos que conduzcan a soluciones buenas o que se encuentren máximos locales, y no globales, como se verá más adelante. Así pues, estas técnicas son propensas a encontrar máximos locales que no son la mejor solución posible.

Los problemas que intentan resolverse con estas técnicas suelen ser *NP-duros*. Dado que dichos problemas requieren (probablemente) al menos un tiempo exponencial para ser

resueltos de manera óptima, cualquier intento práctico de resolverlos pasa por conformarse con la búsqueda de soluciones subóptimas razonablemente buenas. Tal es el caso de los algoritmos de optimización basados en búsqueda local.

En los citados algoritmos, normalmente el camino que se sigue hasta hallar la solución no es importante, razón por la cual no es necesario mantener una estructura que almacene toda la búsqueda realizada hasta el momento (típicamente, un árbol). Otras características de estos métodos son: todos los estados son una solución; normalmente no se espera encontrar la mejor solución, sino una solución que sea lo suficientemente buena; y la tarea de las búsquedas locales es encontrar el mínimo o el máximo de una función, como se adelantó anteriormente.

La representación general de un problema de optimización en el que se aplica búsqueda local pasa por definir:

- Una *representación para los estados*, es decir, la estructura de datos que soportará el estado. Los estados son las *soluciones* posibles del problema.
- Una *función objetivo*, que será la función cuyo valor se trata de optimizar.
- Una *función que genere el estado inicial*. Si en el problema el estado inicial no está claramente definido, entonces éste se podrá generar bien aleatoriamente o bien usando alguna técnica heurística.
- Una *función que genere sucesores* a partir de un estado dado, es decir, las soluciones que son consideradas similares. Define la noción de *vecindad* para el problema concreto y normalmente existe cierta componente aleatoria en ella.

A continuación veremos las principales características de los métodos más relevantes en este área: la escalada, el enfriamiento simulado, los algoritmos genéticos, la optimización basada en nubes de partículas y los algoritmos de colonias de hormigas. Acabamos la sección introduciendo nuestro método heurístico basado en la formación de los ríos.

2.1. Escalada

Este método de búsqueda local se conoce en inglés con el nombre de *hill climbing*. La *escalada* [74, 89, 92] consiste en aplicar técnicas de mejora iterativa para solucionar problemas representados como espacios de estados. Para ello, se confía plenamente en la heurística

definida, pues se escoge en cada paso el sucesor con mejor valor bajo dicha heurística. Uno de los problemas de este método es que no permite recuperarse de un estado erróneo, pues no se mantiene el árbol de búsqueda; además, dependiendo del estado inicial seleccionado, puede quedar *atascado* en un máximo local.

Existen dos tipos de algoritmos de escalada: la escalada simple y la escalada por máxima pendiente.

En la *escalada simple*, se busca aplicar a la solución actual un operador que mejore a ésta. En cuanto encuentre entre los operadores disponibles un operador que mejore la solución, éste se aplicará y se desechará la solución anterior, conservando la nueva.

En la *escalada por máxima pendiente*, se aplican a la solución actual todos los operadores posibles y se actualizará la solución con el movimiento que mejore más a la solución actual (solución *padre*). Entonces se estudia el resultado obtenido con todos los operadores para seleccionar la mejor solución obtenida tras aplicarlos. Esto difiere del método de la escalada simple, en el que cuando se encuentra un operador que mejora la solución actual se deja de probar con el resto.

A continuación se puede ver el *esquema básico* de escalada:

```
algoritmo HillClimbing
   solucionActual = estadoInicial
   fin = falso
   mientras no fin hacer
      hijos = generarSucesores(solucionActual)
      hijos = ordenarYEliminarPeores(hijos, solucionActual)
      si no vacio(hijos) entonces
         solucionActual = escogerMejor(hijos)
      sino
         fin = cierto
      fin si
   fin mientras
fin algoritmo
```

Como se puede observar, este algoritmo representa el algoritmo de escalada por máxima pendiente, pues escoge el mejor de los hijos de la solución actual (*escogerMejor(hijos)*). Por otro lado, préstese atención a que sólo se considerarán aquellos hijos cuya función de

Figura 2.1: Problemas en el espacio de soluciones

estimación sea mejor que la del padre *(ordenarYEliminarPeores(hijos, solucionActual)*).

Las principales ventajas de este método son las siguientes:

- Es muy eficiente en tiempo.
- No mantiene un árbol de búsqueda, con el consiguiente ahorro de memoria.

Se debe tener en cuenta que es posible que este algoritmo no encuentre siempre la solución óptima. En este hecho va a influir en gran medida la calidad de la función heurística definida, que determinará el éxito y la velocidad del algoritmo. Ahora bien, este algoritmo encuentra sus principales *problemas* en los máximos locales (donde no hay ningún vecino con mejor coste), en las mesetas (donde los vecinos son soluciones de igual coste y cuando dejan de serlo tienen peores costes) y en los hombros o crestas (donde los vecinos son soluciones de igual coste y cuando dejan de serlo tienen por un lado costes peores, pero por el otro costes mejores) - Figura 2.1.

Al encontrarse en alguno de estos casos e intentar moverse a una solución vecina, generalmente se empeorará la solución o como mucho no se conseguirá mejorarla. Por tanto, el algoritmo devolverá el estado actual sin ser éste el máximo global del espacio de soluciones. Para evitar estos problemas se pueden aplicar varias medidas:

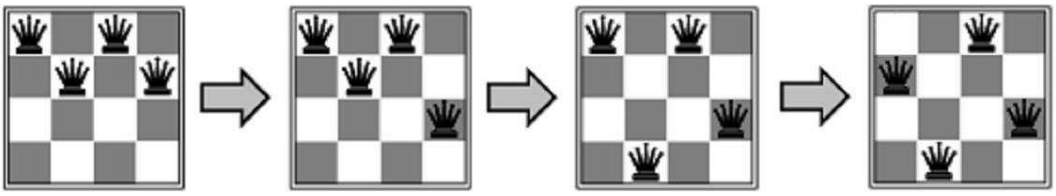

Figura 2.2: Solución al problema de las 4 reinas

- Hacer *backtracking* o vuelta atrás. Esta solución es prohibitiva en espacio, pues habría que guardar multitud de estados. Al ser un espacio de soluciones tan grande, habría tanto problemas de memoria como de tiempo.
- Reiniciar la búsqueda en otro punto. Esta solución es sencilla de implementar y consistiría en empezar la escalada desde otro estado inicial y comprobar si se alcanza una solución mejor.
- Aplicar dos o más operadores antes de decidir el siguiente paso en el camino a seguir (esta medida sólo es aplicable a la escalada simple, pues esta táctica ya se aplica en la escalada por máxima pendiente).
- Hacer *hill climbing* en paralelo, explorando las regiones más prometedoras y consiguiendo así varias soluciones que podrán ser comparadas entre ellas para seleccionar la mejor. Se exploran a la vez N regiones distintas, comparando así los N resultados calculados (a diferencia de reiniciar la búsqueda en otro punto, en la que sólo se compara la nueva solución hallada con el resultado obtenido en la anterior escalada).
- Permitir algunos malos movimientos para así poder escapar de máximos locales (es lo que se hace en el *enfriamiento simulado*, como se verá en la sección posterior).

Un ejemplo clásico al que se ha aplicado la escalada es el *problema de las N reinas*. Este problema consiste en colocar N reinas en un tablero de ajedrez de $N \times N$ casillas de tal forma que no se ataquen entre sí - Figura 2.2 -. En este problema el espacio de búsqueda es de N^N combinaciones. Dos reinas *no se atacan entre sí* si no hay dos reinas en la misma fila, columna o diagonal.

La función heurística puede definirse, por ejemplo, como el número de reinas que se atacan entre sí. En el caso de la Figura 2.3 (a) la función heurística valdría 1, pues hay una única pareja de reinas que se atacan entre sí (la reina de la cuarta columna y la reina de la séptima columna). En el caso de la Figura 2.3 (b) la función heurística valdría 17.

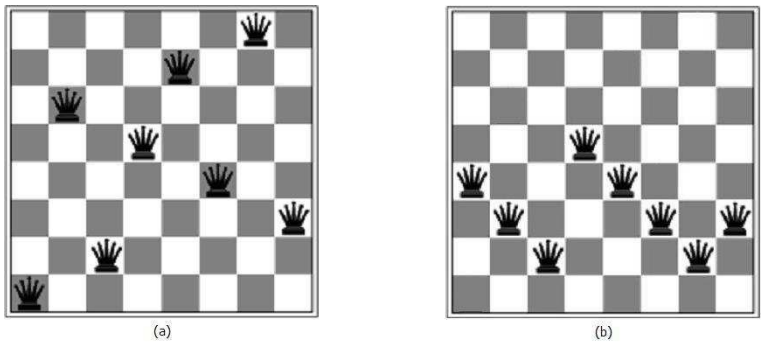

Figura 2.3: Estados en el problema de las 8 reinas

Algunos ejemplos de los operadores que se podrían definir en este problema para la generación de sucesores serían: mover la reina en su columna, lo que generaría $(N \times N) - N$ sucesores, o mover la reina a una casilla contigua de su columna, lo que generaría un máximo de $2 \times N$ sucesores. Por último, el estado inicial del que partiría la escalada podría ser cualquiera. De este modo, quedaría definido el problema de las N reinas para poder aplicar la escalada.

2.2. Enfriamiento Simulado

Este método de búsqueda local se conoce en inglés con el nombre de *simulated annealing* [60, 5, 37, 16]. Consiste en realizar una *escalada* estocástica, inspirada en el enfriamiento controlado de metales, en la cual se calienta un metal a alta temperatura y se enfría progresivamente de manera controlada, lo que se conoce con el nombre de *cristalización* o *templado de metales*. En este proceso, si el enfriamiento ha sido el adecuado, se alcanza una estructura de menor energía que la original, que toma el nombre de *mínimo global*.

En este método se elige un sucesor de entre todos los posibles según una distribución de probabilidad. En dicha elección se permite empeorar la solución actual, es decir, pueden escogerse (probabilísticamente) estados peores. De este modo, se dan pasos parcialmente aleatorios por el espacio de soluciones buscando la mejor solución. Otra de las características de este método es que la probabilidad de que un estado peor sea aceptado varía en función del incremento producido en la función objetivo. Esto permite al algoritmo poder salir de óptimos locales.

A continuación se muestra la metodología de trabajo, donde se identifican los elementos del problema computacional con el problema físico de la siguiente manera:

- La temperatura se va a emplear como parámetro de control. Al principio, la temperatura será elevada para que haya mayor probabilidad de aceptación de soluciones candidatas, lo que se conoce con el nombre de *diversificación*. Al final del proceso, la temperatura será baja, pues se aceptarán pocas soluciones candidatas, lo que se conoce con el nombre de *intensificación*.
- La energía va a ser la función heurística que controle la calidad de la solución ($f'(solucion)$).
- La función que determina el sucesor depende de la temperatura y de la diferencia entre la calidad de los nodos o soluciones ($F(\Delta f', T)$).

Uno de los *objetivos* que se deben conseguir es que, a menor temperatura, haya menor probabilidad de elegir sucesores peores. Para ello, en la estrategia de enfriamiento se debe determinar experimentalmente el número de iteraciones de la búsqueda, la disminución de la temperatura y el número de pasos que se deben dar para cada temperatura. Habrá que ajustar estos parámetros para un buen funcionamiento del algoritmo de enfriamiento simulado. Si la temperatura decrece lo suficientemente despacio, del modo adecuado, entonces se obtendrá la estructura de menor energía consiguiendo un óptimo global. Así, el *criterio de parada* puede venir dado por haber conseguido un valor suficientemente bueno de la función objetivo, o por haber dado un número determinado máximo de iteraciones sin conseguir ninguna mejora en la solución, o por haber alcanzado el número de iteraciones totales.

A continuación se puede ver el *esquema básico* del enfriamiento simulado:

```
algoritmo SimulatedAnnealing
   solucionActual = estadoInicial
   t = 1
   fin = false
   mientras (t <= numeroDeVueltas) y (no fin) hacer
      temperatura = planificarValor(t)
      si temperatura = 0 entonces
         fin = true
      sino
         nodoSiguiente = elegirSucesorAleatorio(solucionActual)
         incrementoEnergia =
            F(f'(solucionActual) - f'(nodoSiguiente), temperatura)
```

```
            si incrementoEnergia > 0 entonces
               solucionActual = nodoSiguiente
            sino
               p = valorAleatorioEntre0y1()
               si p <= ciertaProbabilidad entonces
                  solucionActual = nodoSiguiente
               fin si
            fin si
         fin si
      fin mientras
      devolver solucionActual
   fin algoritmo
```

Un ejemplo de búsqueda con *enfriamiento simulado* puede verse en la Figura 2.4, en la que se ilustra cómo en ciertos puntos de la búsqueda se elige una solución peor que la actual para poder huir de los máximos locales y poder encontrar el máximo global (*estado final*). En el gráfico, las regiones coloreadas más oscuras representan soluciones mejores, mientras que las regiones coloreadas más claras representan soluciones peores.

Esta técnica está indicada para problemas grandes donde el óptimo global está rodeado de óptimos locales y para problemas en los que es difícil encontrar una heurística discriminante (es decir, una heurística que conduzca eficientemente a la solución).

El principal problema del enfriamiento simulado es determinar los valores de los parámetros, lo que requiere de un proceso de experimentación.

Veamos cómo se podría aplicar el *enfriamiento simulado* al problema de *el viajante de comercio* (en adelante TSP, del inglés *Traveling Salesman Problem* [42, 8]). Muy brevemente, el TSP consiste en que, dado un número de ciudades y los costes de viajar de una ciudad a otra, debe calcularse el recorrido más barato que visita cada ciudad exactamente una vez y finaliza el recorrido en la ciudad inicial. A continuación veremos cómo se puede aplicar el enfriamiento simulado a la resolución de este problema. El espacio de búsqueda en este problema es $N!$ y como operadores se podrían definir la traslación y el intercambio. La *traslación* consiste en que a partir de una solución, por ejemplo *A-B-C-D-E-A*, se genera una nueva trasladando una ciudad visitada a otra posición. En el ejemplo, si se traslada la ciudad B a ser la cuarta ciudad visitada, resultaría *A-C-D-B-E-A*. El operador de *intercambio* consiste

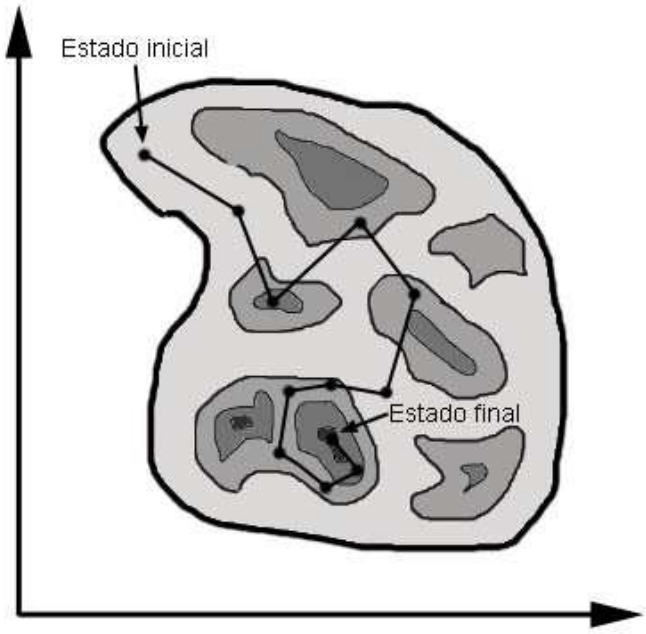

Figura 2.4: Búsqueda en el espacio de soluciones

en cambiar de posición a una ciudad visitada por otra. En el ejemplo, si se intercambian las ciudades B y D se obtiene A-D-C-B-E-A. La función de energía que se define para resolver el problema es la suma de la distancia entre las ciudades según el orden de la solución. Definir la temperatura inicial depende de la experimentación, al igual que el número de iteraciones para cada una de las temperaturas y cómo debe disminuir la temperatura.

2.3. Algoritmos Genéticos

Este tipo de métodos de búsqueda local [41, 25, 72, 44] consiste en realizar una *escalada* en paralelo inspirada en los siguientes mecanismos de selección natural:

- La adaptación de los seres vivos al entorno en el que viven.
- La supervivencia y reproducción y sus posibilidades según las características de cada individuo.
- La combinación de individuos, que puede llevar a individuos mejor adaptados.

Las analogías de la selección natural con la búsqueda local son las siguientes:

- Las posibles soluciones se representan como los seres vivos, es decir, como los individuos que se adaptan al medio.
- La función que evalúa la calidad de cada solución calcula la adaptación de cada individuo.
- Basándose en la reproducción de los individuos, se supone que combinando buenas soluciones se pueden obtener soluciones mejores.
- Apoyándose en el mecanismo de evolución, se seleccionan las mejores soluciones sucesivamente.

El esquema a seguir para resolver un problema utilizando algoritmos genéticos es el siguiente:

- El primer paso consiste en *codificar* las características de las soluciones, por ejemplo mediante una cadena binaria.
- A continuación se debe definir una función de calidad de la solución, que en inglés recibe el nombre de función de *fitness*.
- Para generar las nuevas soluciones se definen dos *operadores*: el cruzamiento (*crossover*) y la mutación (*mutation*).
- El siguiente paso consiste en decidir el número de *individuos iniciales*.
- Por último, queda decidir la estrategia a seguir para la combinación de individuos.

Respecto a la codificación de individuos, habitualmente se realiza usando una *cadena binaria*. De este modo, en el problema de las N reinas podrían definirse los individuos como aparece en la Figura 2.5. Esta codificación $[00, 10, 01, 11]$ define la posición de las 4 reinas del tablero. Así, la reina de la fila 1 está en la columna 1 (representada por 00); la reina de la fila 2 está en la columna 3 (representada por 10); la reina de la fila 3 está en la columna 2 (representada por 01); y la reina de la fila 4 está en la columna 4 (representada por 11). Se utiliza esta representación ya que, en una solución del problema de las N reinas, no puede haber dos reinas en la misma fila (o columna). La codificación utilizada para representar a los individuos determina el tamaño del espacio de búsqueda y el tipo de operadores de combinación necesarios.

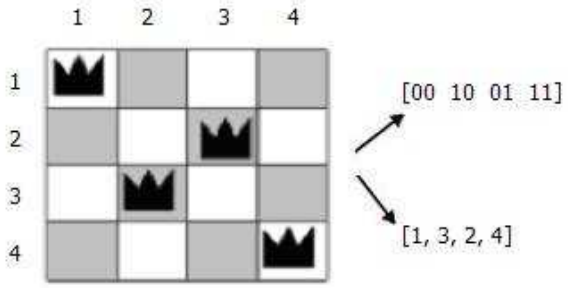

Figura 2.5: Codificación de individuos en el problema de las 4 reinas

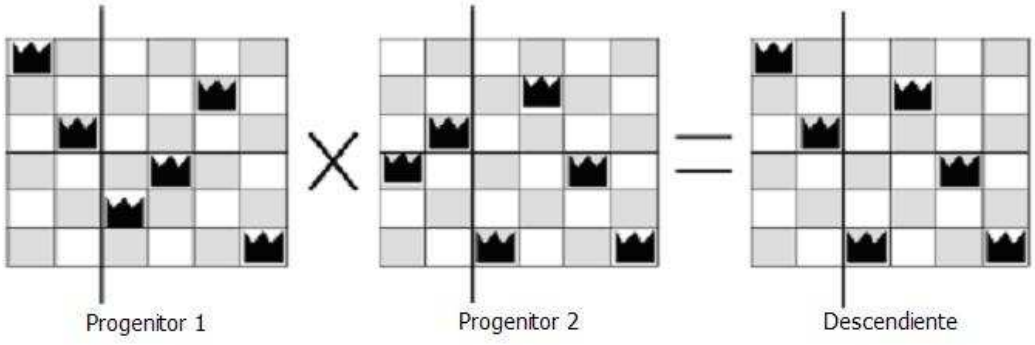

Figura 2.6: Funcionamiento del operador de cruce

Un operador básico en los algoritmos genéticos es el *cruce por un punto*. En el siguiente ejemplo - Figura 2.6 - se puede observar cómo funciona este operador. Este operador mezcla o cruza dos individuos o *progenitores* para obtener un nuevo individuo, el *descendiente*. En este ejemplo el descendiente está formado por la parte izquierda del progenitor 1 y la parte derecha del progenitor 2. Igualmente, se podría crear otro descendiente mezclando la parte derecha del progenitor 1 con la parte izquierda del progenitor 2. La línea negra más gruesa que atraviesa el tablero de ajedrez del ejemplo verticalmente es el *punto de cruce*. Este punto de cruce se elige de forma aleatoria.

Otras posibilidades para definir el cruce son el *cruce en dos puntos* - en el cual existen dos puntos de cruce - y el *intercambio aleatorio de bits*.

El otro operador utilizado en los algoritmos genéticos es la *mutación*, que consiste en cambiar el valor de un bit con cierta probabilidad. Esta probabilidad debe calcularse expe-

rimentalmente.

En estos algoritmos, en cada paso de la búsqueda el estado queda determinado por un conjunto de N individuos (una *generación*), siendo N constante. Para pasar a la siguiente generación (siguiente paso) se deben elegir los individuos que se van a combinar.

Para realizar la elección de los individuos que van a ser progenitores, se asocia una probabilidad proporcional a la calidad del individuo; a continuación se realizan N enfrentamientos aleatorios entre parejas de individuos y se eligen a los que ganan; por último, se define un *ranking* de individuos según su calidad. Es posible que haya individuos que aparezcan más de una vez, mientras que habrá otros individuos que no aparezcan.

A continuación se puede ver el *esquema básico* de los algoritmos genéticos:

```
algoritmo GeneticAlgorithm
   i = 0
   poblacionInicial = crearPoblacionInicial()
   generacionFinal = poblacionInicial
   mientras (no converge(generacionFinal)) y (i < numIteraciones) hacer
      generacionIntermedia = escogerNIndividuos(poblacionInicial)
      generacionCruce =
         aplicarCruce(generacionIntermedia, probabilidadCruce)
      generacionMutacion =
         aplicarMutacion(generacionCruce, probabilidadMutacion)
      generacionFinal =
         escogerMejores(generacionFinal, generacionMutacion)
      i = i + 1
   fin mientras
fin algoritmo
```

Los parámetros del algoritmo genético son:

- El número total de individuos de la población (que es constante en cada generación).
- La proporción del total de los individuos que intervienen en la reproducción en cada generación, es decir, la proporción de individuos que serán padres.

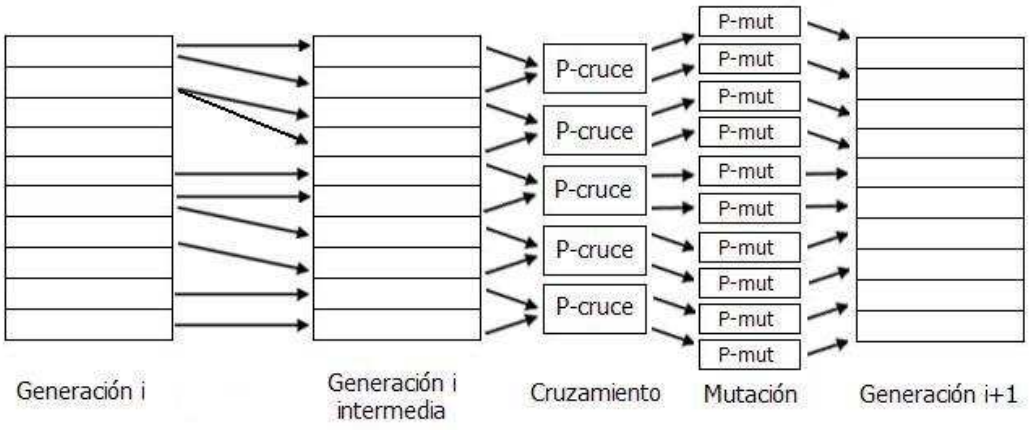

Figura 2.7: Pasos de un algoritmo genético (1)

- La proporción del total de los padres que se escogen de entre los mejores individuos de la población, es decir, la proporción de padres que se eligen por su *ranking* dentro de la población.
- La probabilidad de que ocurra una mutación, que generalmente es un número muy bajo.

En la selección de los padres, un número fijo de ellos se obtienen seleccionando los mejores de la población, es decir, aquellos que son más prometedores. Los restantes se eligen aleatoriamente de entre los demás, para mantener así la *diversidad*. A la hora de realizar los cruces, las parejas se obtienen reordenando aleatoriamente a los padres y cruzándolos de dos en dos. Una vez realizados los cruces, cada característica (es decir, cada *gen*) de cada individuo resultante se mutará según la probabilidad dada. Por último, para obtener la nueva generación, se unen la generación anterior y la nueva para seleccionar a los mejores individuos (en igual número que la generación inicial). En la Figura 2.7 se puede observar cómo varía la población en cada uno de los pasos del algoritmo.

Se debe tener en cuenta que existen otras posibles implementaciones de los algoritmos genéticos, pero todas se basan en las mismas ideas de reproducción, mutación, selección de los mejores y mantenimiento de la diversidad.

En la Figura 2.8, se puede ver cómo serían los pasos del esquema de un algoritmo genético en un ejemplo y cómo un individuo puede ser seleccionado dos veces mientras otro puede no ser elegido.

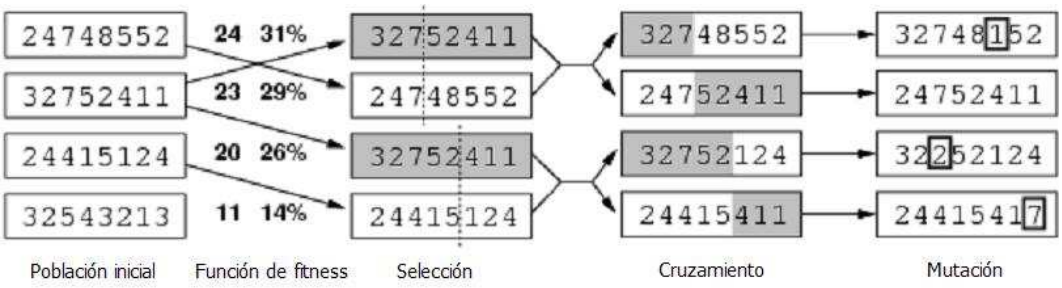

Figura 2.8: Pasos de un algoritmo genético (2)

En este caso, hay una variante en la selección de los individuos para ser padres, ya que cada individuo i es seleccionado para ser padre con probabilidad proporcional a su valor de la función objetivo $F_{obj}(i)$, lo que se conoce con el nombre de *ruleta probabilística*:

$$P(i) = \frac{F_{obj}(i)}{\sum_{j=1}^{N} F_{obj}(j)}$$

Los cruces, las mutaciones y la diversidad tratan de evitar el problema de los óptimos locales, aunque las mayores ventajas asociadas a este tipo de algoritmos son que se pueden aplicar a cualquier tipo de problema (optimización, aprendizaje automático, planificación, etc.) y que permite abordar problemas para los que no se dispone de una heurística adecuada, aunque en general serán peores que un algoritmo clásico con una heurística bien definida.

Sin embargo, también presentan varios problemas, como que la codificación de los estados no es ni fácil ni intuitiva y que es complicado definir los parámetros del algoritmo de manera adecuada (el tamaño de la población, el número de iteraciones y las probabilidades de cruce y de mutación).

Los algoritmos genéticos se pueden aplicar al problema de las N reinas del siguiente modo: cada posible solución se puede codificar como una cadena binaria, en la que cada individuo se representa por la concatenación de la fila y la columna en la que se encuentra la reina de dicha fila, es decir, $Concat(i = 1..N; Binario(Columna(reina_i)))$. Por otro lado, la función de fitness podría definirse como el número de parejas de reinas que se atacan entre sí. Además, como operador de cruce podría utilizarse el cruce en un punto de forma que la selección de la generación intermedia fuese proporcional a la función de fitness. Por último, la probabilidad de cruce, la probabilidad de mutación y el tamaño de la población inicial, se tendrían que definir experimentalmente para conseguir los mejores resultados posibles.

2.4. Optimización Basada en Nubes de Partículas

Este método se conoce en inglés con el nombre de *particle swarm optimization* (en adelante PSO). PSO [58, 21, 76] es una metaheurística poblacional inspirada en el comportamiento social del vuelo de las bandadas de aves y el movimiento de los bancos de peces. En estos sistemas, la población se compone de varias partículas (de ahí su nombre) que se mueven (o *vuelan* en el caso de las bandadas de aves o de enjambres de insectos) por el espacio de búsqueda durante la ejecución del algoritmo. Estas entidades son muy simples y tienen interacciones locales (incluyendo interacciones con el ambiente). El resultado de la combinación de comportamientos simples es la aparición de comportamientos complejos y la capacidad de conseguir buenos resultados como un *equipo*.

Una disciplina del PSO es la tecnología basada en nubes de partículas *inteligentes*, que está basada en la tecnología de nubes de partículas. A los miembros individuales del *enjambre* se les añade inteligencia como seres independientes. Con estos *enjambres inteligentes*, los miembros pueden ser homogéneos o heterogéneos. Incluso si los miembros comienzan siendo homogéneos, pueden aprender cosas diferentes, debido a las diferencias del medio, desarrollar diferentes objetivos y entonces convertirse en un grupo heterogéneo. Las técnicas de nubes de partículas con inteligencia son métodos estocásticos basados en población usados en problemas de optimización combinatoria, donde el comportamiento colectivo de los individuos surge de sus interacciones locales con el medio para alzar la aparición de patrones globales funcionales. También se han utilizado las nubes de partículas donde los agentes son dispositivos robots físicos.

El movimiento de cada una de las partículas p va a depender de:

- Su mejor posición desde que comenzó el algoritmo, *pBest*.
- La mejor posición de las partículas de su entorno, *IBest*, o de toda la nube de partículas, *gBest*, desde que comenzó el algoritmo.

La velocidad de p se cambiará aleatoriamente en cada iteración del algoritmo para acercarla a las posiciones *pBest* e *IBest/gBest*.

Este método se aplica típicamente en problemas de optimización numérica y se le atribuyen las siguientes características:

- Asume un intercambio de información entre los agentes de búsqueda, lo que se identifica con las *interacciones sociales* de las aves o de los peces.

- La idea básica es utilizar la información del mejor resultado propio de cada partícula y la información del mejor resultado global.
- La implementación es muy sencilla ya que se manejan pocos parámetros.
- La convergencia del algoritmo es rápida y se encuentran buenas soluciones.

Veamos el funcionamiento básico del PSO. Como se ha visto anteriormente, una nube de partículas simula el comportamiento de las bandadas de aves. En particular, se supone que una de estas bandadas busca comida en un área y que solamente hay una pieza de comida en dicha área. Se asume que los pájaros de la bandada no saben dónde está la comida, pero sí conocen su distancia a la misma, por lo que la estrategia más eficaz para hallar la comida será seguir al ave que se encuentre más cerca de ella.

El PSO emula este escenario para resolver problemas de optimización. Cada solución o *partícula* es un *ave* en el espacio de búsqueda que está siempre en continuo movimiento y que nunca muere.

De hecho, se puede considerar que la nube de partículas es un *sistema multiagente* en el que las partículas son agentes simples que se mueven por el espacio de búsqueda y que guardan (y posiblemente comunican) la mejor solución que han encontrado.

Cada partícula tiene un valor de *fitness*, una posición y un vector velocidad que dirige su *vuelo*. El movimiento de las partículas por el espacio de búsqueda es guiado por las partículas óptimas en el momento actual. Una partícula está compuesta por:

- Tres vectores:
  - El *vector X*, que almacena la posición actual (la localización) de la partícula en el espacio de búsqueda.
  - El *vector pBest*, que almacena la localización de la mejor solución encontrada por la partícula hasta el momento.
  - El *vector V*, que almacena el gradiente (la dirección) según el cual se moverá la partícula.
- Dos valores de fitness:
  - El *x_fitness*, que almacena el fitness de la solución actual (*vector X*).
  - El *p_fitness*, que almacena el fitness de la mejor solución local (*vector pBest*).

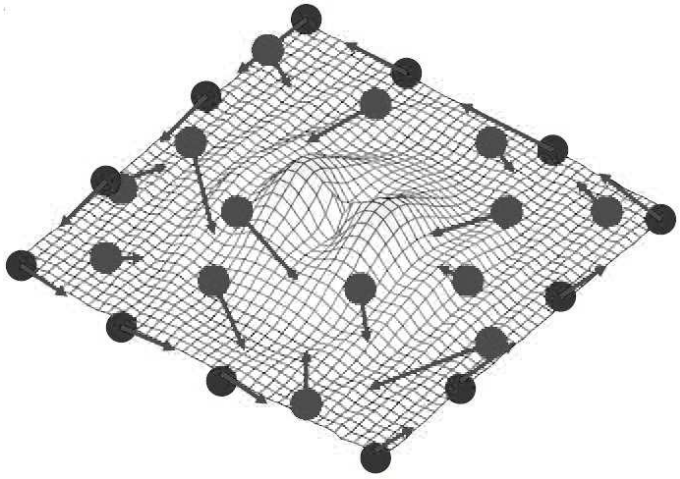

Figura 2.9: Posiciones y velocidades iniciales de las partículas

Para la inicialización de la nube de partículas, se tienen que generar las posiciones y las velocidades iniciales de las partículas. Las *posiciones* se pueden generar aleatoriamente en el espacio de búsqueda, de forma regular, o por medio de una combinación de ambas. Las *velocidades* se generan aleatoriamente con cada componente en el intervalo $[-V_{max}, V_{max}]$ -Figura 2.9 -, aunque no es conveniente fijarlas a cero, pues no se obtienen buenos resultados. V_{max} será la velocidad máxima que pueda tomar una partícula en cada movimiento.

Ahora se verá cómo se mueve una partícula de una posición del espacio de búsqueda a otra. Para ello, simplemente se añade el vector velocidad V_i al vector posición X_i, para obtener así un nuevo vector posición: $X_i \leftarrow X_i + V_i$.

Una vez se ha calculado la nueva posición se evaluará ésta y si el nuevo fitness resulta ser mejor que el que la partícula tenía hasta ahora, *pBest_fitness*, entonces:

$$pBest_i \leftarrow X_i \ ; \ pBest\_fitness \leftarrow x_{fitness}$$

.

En cada paso, las fórmulas utilizadas para actualizar el vector velocidad de una partícula p_i y su posición son las siguientes, donde d es la d-ésima dimensión de los vectores:

$$v_{i,d} = \omega \cdot v_{i,d} + \varphi_1 \cdot rnd() \cdot (pBest_{i,d} - x_{i,d}) + \varphi_2 \cdot rnd() \cdot (g_{i,d} - x_{i,d})$$

$$x_{i,d} = x_{i,d} + v_{i,d}$$

donde:

- ω representa una constante de inercia. Normalmente toma valores ligeramente menores que 1.
- φ_1 y φ_2 son ratios de aprendizaje (pesos) que controlan los componentes *cognitivo* y *social*. Se conoce con el nombre de *componente cognitivo* a la expresión: $\varphi_1 \cdot rnd() \cdot (pBest_{i,d} - x_{i,d})$; y con el nombre de *componente social* a la expresión: $\varphi_2 \cdot rnd() \cdot (g_{i,d} - x_{i,d})$
- g representa el índice de la partícula con el mejor *pBest_fitness* del entorno de p_i (*IBest*) o de toda la nube (*gBest*).
- *rnd()* genera un número aleatorio en el intervalo [0,1).

Kennedy [58] identifica tres tipos de algoritmos de PSO en función de los valores φ_1 y φ_2:

- El *modelo completo*, en el que φ_1, $\varphi_2 > 0$.
- El *modelo sólo cognitivo*, en el que $\varphi_1 > 0$ y $\varphi_2 = 0$.
- El *modelo sólo social*, en el que $\varphi_1 = 0$ y $\varphi_2 > 0$.

En la *Figura 2.10* se puede ver la representación gráfica del movimiento de las partículas.

A continuación se puede ver el pseudocódigo del *PSO*:

```
algoritmo PSO
   t = 0
   desde i = 1 a numeroDeParticulas hacer
      inicializar X_i y V_i
   fin desde
   mientras no criterioDeParada() hacer
      t = t + 1
      desde i = 1 a numeroDeParticulas hacer
         evaluar X_i
         si F(X_i) es mejor que F(pBest) entonces
            pBest_i = X_i  (F(pBest_i) = F(X_i))
         fin si
```

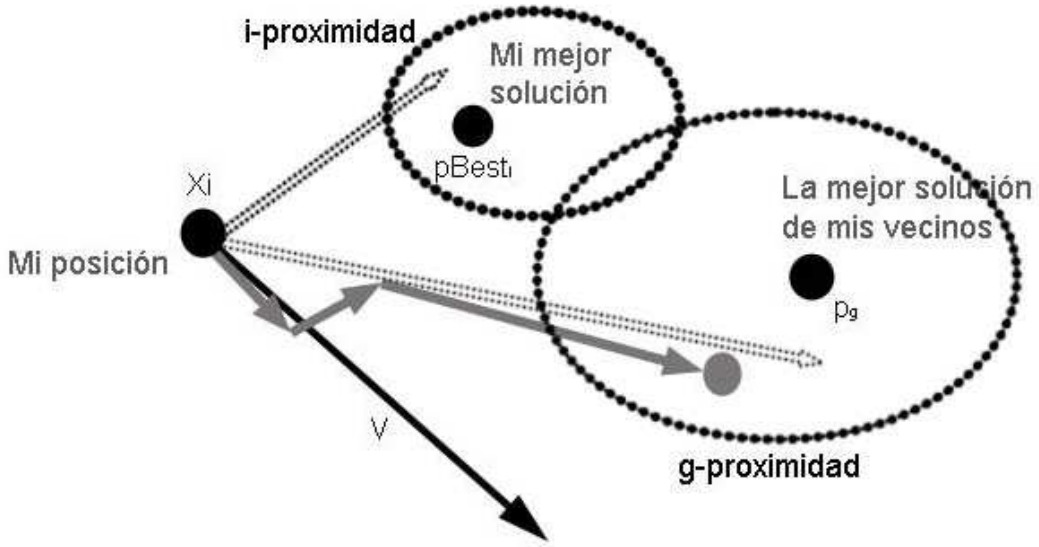

Figura 2.10: Representación gráfica del movimiento de una partícula

```
        fin desde
        desde i = 1 a numeroDeParticulas hacer
            Escoger IBest_i, la partícula con mejor fitness del entorno de X_i
            Calcular V_i, la velocidad de X_i, de acuerdo a pBest_i y IBest_i
            Calcular la nueva posición X_i, de acuerdo a X_i y V_i
        fin desde
    fin mientras
    Devolver la mejor solución encontrada
fin algoritmo
```

Otra versión del algoritmo es el *PSO global*, muy similar al anterior, que se muestra a continuación:

```
algoritmo PSOGlobal
    t = 0
    desde i = 1 a numeroDeParticulas hacer
        inicializar X_i y V_i
    fin desde
    mientras no criterioDeParada() hacer
        t = t + 1
```

```
        desde i = 1 a numeroDeParticulas hacer
            evaluar X_i
            si F(X_i) es mejor que F(pBest) entonces
                pBest_i = X_i  (F(pBest_i) = F(X_i))
            fin si
            si F(pBest) es mejor que F(gBest) entonces
                gBest = pBest_i  (F(gBest_i) = F(pBest_i))
            fin si
        fin desde
        desde i = 1 a numeroDeParticulas hacer
            Calcular V_i, la velocidad de X_i, de acuerdo a pBest_i y gBest_i
            Calcular la nueva posición X_i, de acuerdo a X_i y V_i
        fin desde
    fin mientras
    Devolver la mejor solución encontrada
fin algoritmo
```

Si se compara el PSO global y el anterior algoritmo del PSO, la versión global converge más rápidamente, pero tiene el inconveniente de que cae más fácilmente en óptimos locales.

Las nubes de partículas pueden formar distintas *topologías*. Estas topologías definen el entorno de cada partícula individual (se asume que cada partícula siempre pertenece a su propio entorno). Estos entornos - Figura 2.11 - pueden ser de dos tipos:

- *Geográficos*: en los que se calcula la distancia de la partícula actual al resto y se toman las más cercanas para componer su entorno.
- *Sociales*: en los que se define *a priori* una lista de vecinas para cada partícula, independientemente de su posición en el espacio.

Generalmente, los entornos sociales son los más empleados. Una vez decidido el tipo de entorno, es necesario definir su tamaño. Lo habitual es definir el tamaño entre 3 y 5 pues dan un buen comportamiento, si bien el algoritmo no es muy sensible a este parámetro. Cuando el tamaño es toda la nube de partículas, el entorno es a la vez geográfico y social, y se tendría el PSO global.

Por otra parte, una de las topologías sociales más empleada es la de anillo, en la que se considera un vecindario circular. Se numera cada partícula, se construye un círculo virtual

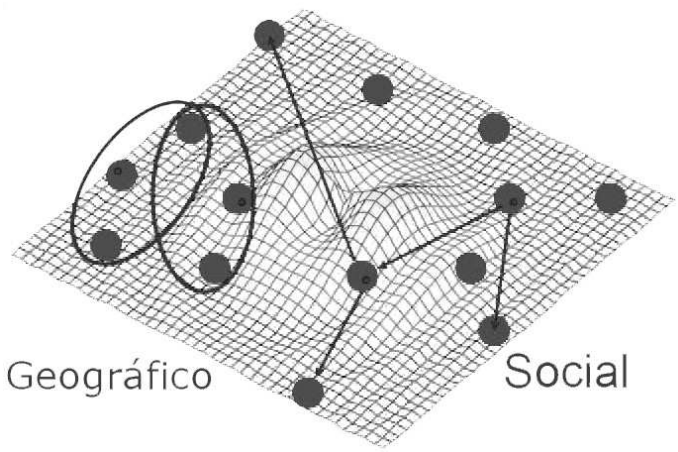

Figura 2.11: Tipos de entornos

con estos números y se define el entorno de una partícula con sus vecinas en el círculo - Figura 2.12 -.

El uso de tecnologías basadas en nubes de partículas se ha convertido en algo común en una variedad de dominios de aplicación: medicina, bioinformática, aplicaciones militares y de defensa, vigilancia, incluso en la transmisión de televisión e internet. La idea de que estos métodos se pueden usar para resolver problemas complejos ha sido utilizada en muchas áreas diferentes de la informática. De este modo, el PSO ha sido aplicado a numerosos problemas como la optimización de funciones numéricas, el entrenamiento de redes neuronales, el aprendizaje de sistemas difusos, el problema del viajante de comercio, etc.

2.5. Algoritmos de Colonias de Hormigas

Este método se conoce en inglés con el nombre de *ant colony optimization* (ACO) y es una técnica probabilística para resolver problemas computacionales que se pueden reducir al problema de encontrar buenos caminos a través de grafos [34, 33, 30, 31, 65]. Se inspira en el comportamiento de las hormigas, que son insectos sociales que viven en colonias y que tienen un comportamiento dirigido al desarrollo de la colonia como un todo más que a un desarrollo individual. Este método se centra en un característica interesante del comportamiento de las hormigas: la manera en que encuentran los caminos más cortos desde el nido u hormiguero hasta la comida, teniendo en cuenta que estos insectos son ciegos.

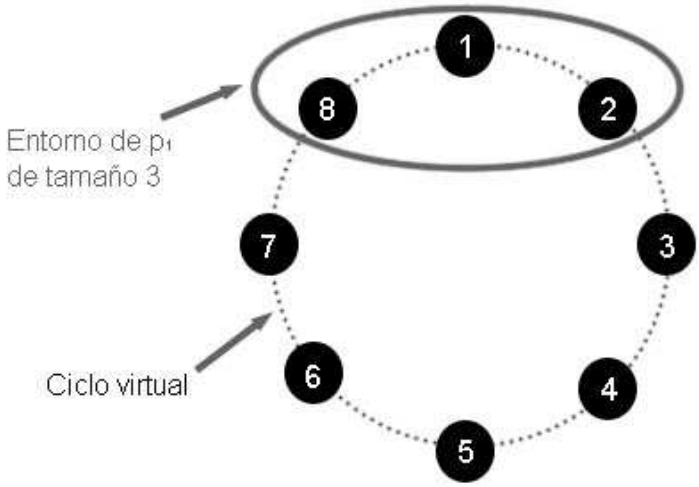

Figura 2.12: Topología en anillo

En el mundo real, las hormigas descubren el camino más corto hasta la comida dejando rastros de *feromonas* que todas pueden oler. Además, este rastro permite a las hormigas volver a su hormiguero desde la comida. El comportamiento básico de las hormigas es el siguiente: al principio cada hormiga se mueve de manera aleatoria; depositan feromonas en el camino que recorren; las hormigas detectan el camino principal (el que tiene más feromonas) y se inclinan a seguirlo, ya que cuantas más feromonas tiene un camino, más probable es que una hormiga siga ese camino.

En la Figura 2.13 se puede observar cómo todas las hormigas de una colonia tienden a seguir el mismo camino para ir desde su nido hasta la comida. Si las hormigas se encuentran un obstáculo en su camino, éstas buscarán por donde evitarlo más rápidamente y finalmente todas seguirán la ruta más corta hasta la comida para poder llevarla a su nido más rápidamente. Esto es debido a que por el camino más corto pasarán las hormigas más rápido y más veces, por lo que se alimenta más el rastro de feromonas por el camino más corto que por el más largo, haciendo así que, posteriormente, más hormigas elijan el camino más corto.

A continuación se puede ver el algoritmo básico ACO:

```
algoritmo AntColonyOptimization
   inicializarRastros()
   mientras no criterioDeParada() hacer
```

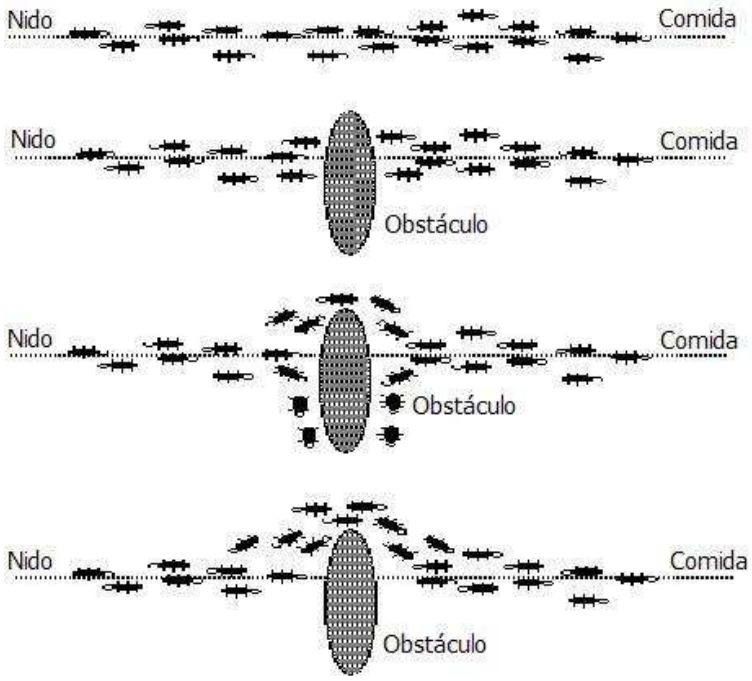

Figura 2.13: Comportamiento de las hormigas ante un obstáculo

```
        hacerRecorridos()
        analizarRutas()
        actualizarRastro()
    fin mientras
fin algoritmo
```

Las dificultades que plantea el método son que el algoritmo se basa en una serie de decisiones aleatorias guiadas por hormigas artificiales y que las probabilidades para tomar las decisiones cambian en cada iteración.

El método ACO se basa en última instancia en las pruebas realizadas por *Deneubourg* [27] y su equipo, quienes realizaron un experimento de laboratorio con un tipo concreto de hormigas que depositan feromonas al ir del hormiguero a la comida y al volver. En su experimento, usaron dos tipos de circuitos para las hormigas, donde en el *primero* las dos ramas del circuito que conducían del nido a la comida tenían la misma longitud y donde en el *segundo* una rama era el doble de larga que la otra. Se realizó un *tercer* experimento en el que se concatenaron dos circuitos del segundo tipo para ver cómo se comportaban las

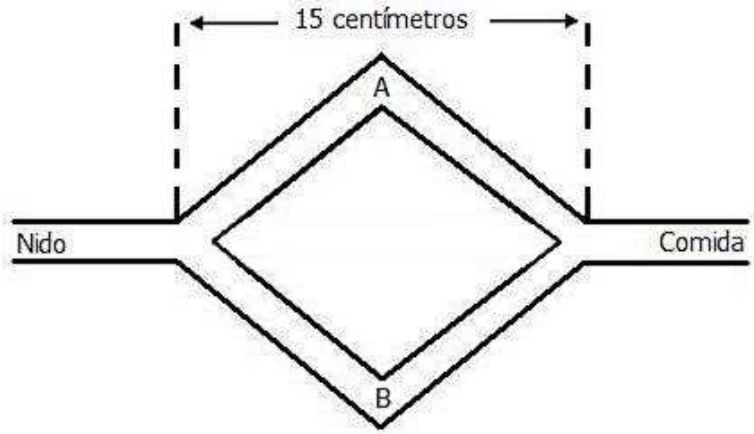

Figura 2.14: Primer circuito del experimento de Deneubourg

hormigas.

En el primer circuito - Figura 2.14 -, las hormigas terminaban por converger a una sola rama, cualquiera de las dos. *Deneubourg* observó que en el segundo circuito - Figura 2.15 -, las hormigas convergían rápidamente a la rama más corta. Finalmente, en el tercer circuito - Figura 2.16 -, con dos zonas como las del anterior experimento concatenadas, las hormigas también consiguen encontrar el camino más corto. Esto es debido a que, dado que el camino corto se recorre antes, este será recorrido más veces por las hormigas que el camino más largo, depositando de esta manera una mayor cantidad de feromonas.

Como resultado de estos experimentos, *Deneubourg* y su equipo diseñaron un modelo estocástico del proceso de decisión de las hormigas naturales:

$$p_{i,a} = \frac{[k+\tau_{i,a}]^{\alpha}}{[k+\tau_{i,a}]^{\alpha}+[k+\tau_{i,a'}]^{\alpha}}$$

donde $p_{i,a}$ es la probabilidad de escoger la rama a estando en el punto de decisión i, $\tau_{i,a}$ es la concentración de feromona en la rama a, a' es otra rama y k y α son constantes.

Los algoritmos de optimización basados en colonias de hormigas se usan típicamente para resolver problemas de minimización del coste de caminos en grafos. En adelante supondremos un grafo de N nodos con A arcos no dirigidos. Existen dos modos de trabajo para las hormigas: hacia delante y hacia atrás. En el modo *hacia delante* se recorre el grafo evitando los ciclos y en el modo *hacia atrás* sólo se depositan las feromonas.

Cada hormiga artificial es un mecanismo probabilístico de construcción de soluciones al

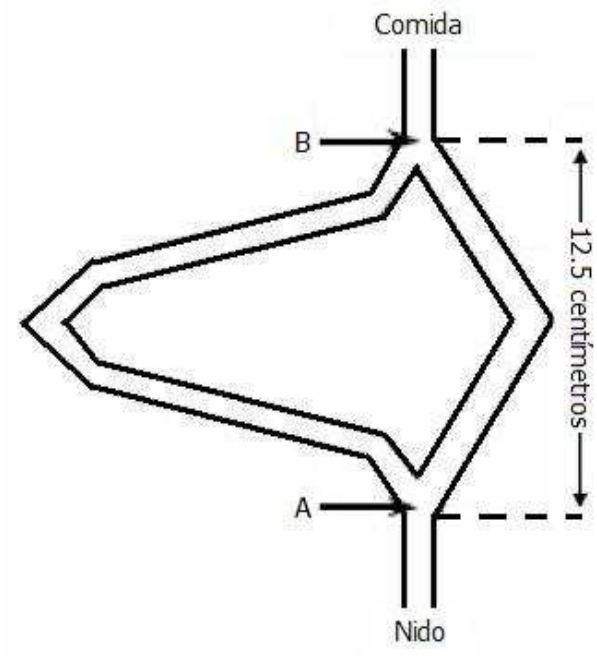

Figura 2.15: Segundo circuito del experimento de Deneubourg

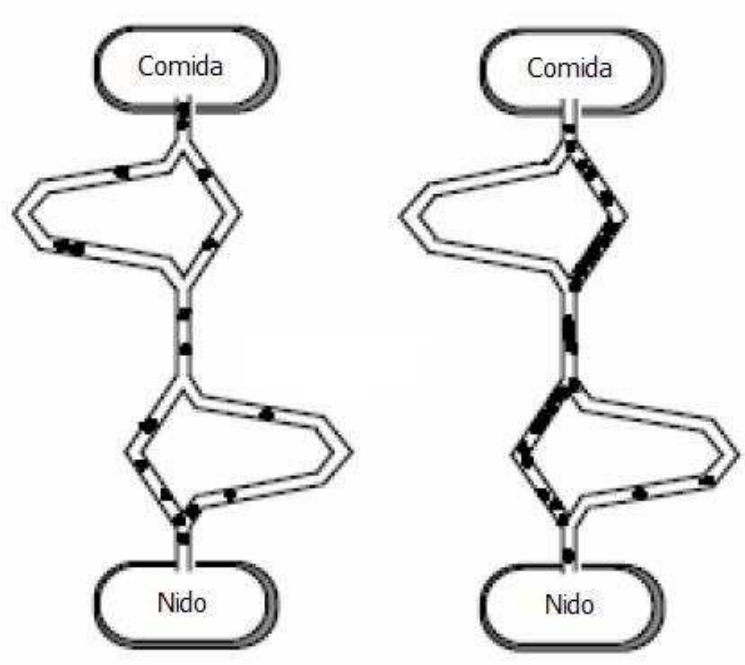

Figura 2.16: Tercer circuito del experimento de Deneubourg

problema (un agente que imita a la hormiga natural). Las hormigas disponen de una *memoria* que les permite recordar el camino que han seguido mientras buscaban el nodo de destino para deshacerlo posteriormente. Antes de volver hacia atrás en su camino memorizado, eliminan cualquier bucle del camino memorizado y cuando vuelven hacia atrás dejan las feromonas en los arcos que atraviesan. Esa memoria no es más que una lista de los nodos visitados, y al finalizar esta lista contiene la solución construida por la hormiga.

Las hormigas son capaces de evaluar el coste de los caminos que han atravesado, para así poder depositar una mayor cantidad de feromonas en los caminos más cortos. Es decir, se depositará una mayor cantidad de feromonas en aquellas soluciones cuyo coste sea más bajo. También se ha de aplicar una regla de evaporación de las feromonas, que reducirá progresivamente la probabilidad de elegir soluciones de baja calidad.

Al principio del proceso de búsqueda, se asigna una cantidad constante de feromonas a todos los arcos del grafo. Cuando una hormiga k se encuentra en el nodo i, usa el rastro de feromonas para calcular la probabilidad de elegir a j como el próximo nodo de su ruta. De esta manera, la probabilidad de que la hormiga k vaya del nodo i al nodo j viene dada por la expresión:

$$p_{ij}^k = \begin{cases} \frac{\omega_{ij}}{\sum_{l \in N_i^k} \omega_{il}} & si \quad j \in N_i^k \\ 0 & si \quad j \notin N_i^k \end{cases}$$

donde N_i^k es el conjunto de vecinos de la hormiga k cuando está en el nodo i y ω_{ij} es la cantidad de feromonas presentes en la arista que va del nodo i al nodo j.

Cuando el arco (i, j) es atravesado, la cantidad de feromonas de ese arco se incrementa en la cantidad de feromonas que ha dejado la hormiga k al pasar por él.

$$\omega_{ij} \leftarrow \omega_{ij} + feromonas_{ij}^k$$

Usando esta regla, se consigue aumentar la probabilidad de que las hormigas que vengan detrás usen este arco. Una vez que todas las hormigas se han movido al siguiente nodo, las feromonas se evaporan de los arcos según la siguiente ecuación:

$$\omega_{ij} \leftarrow (1 - p) \times \omega_{ij} \quad \forall (i, j) \in A$$

donde A es el conjunto de aristas del grafo y $p \in (0, 1]$ es un parámetro. Se usa la evaporación de la feromona para evitar un incremento ilimitado de los rastros de feromona y para permitir olvidar las malas decisiones tomadas. Es un mecanismo de evaporación más activo que el

natural, lo que evita la perduración de los rastros de feromona y, por tanto, el estancamiento en óptimos locales.

Una iteración es un ciclo completo en el que se incluye: el movimiento de las hormigas, la evaporación de feromonas y el depósito de feromonas.

Los pasos que se deben seguir para resolver un problema utilizando ACO son los siguientes:

- Representar el problema como un grafo valorado en el que las hormigas podrán construir soluciones.
- Definir el significado de los rastros de feromonas.
- Definir la heurística que va a permitir a la hormiga construir una solución.
- Si es posible, implementar un algoritmo de búsqueda local eficiente para resolver el problema (para poder comparar los resultados que se van obteniendo con una solución inicial).
- Elegir un algoritmo ACO específico y aplicarlo al problema a resolver.
- Ajustar los parámetros del algoritmo ACO.

Existen muchas extensiones al algoritmo básico creado inicialmente por *Dorigo*, donde, por ejemplo, en la *regla de transición* se establece un equilibrio entre la exploración de nuevos arcos y la explotación de la información acumulada; o donde para la *actualización global de feromona* sólo se considera la hormiga que generó la mejor solución hasta el momento; o donde se añade una nueva *actualización local de feromona* basada en que cada hormiga modifica automáticamente la feromona de cada arco para diversificar la búsqueda.

Otro de los algoritmos de este tipo es el *Sistema de Hormigas Max-Min* [96, 97], que es una nueva extensión con una mayor explotación de las mejores soluciones y un mecanismo adicional para evitar el estancamiento de la búsqueda.

Este sistema mantiene las reglas de transición del algoritmo básico pero realiza las siguientes modificaciones:

- El *mecanismo de actualización* es más agresivo, al evaporar todos los rastros y sólo aportar feromonas en los arcos de la mejor solución.

- Define unos *límites máximos y mínimos para los rastros de feromonas* $\tau_{min} \leq \tau \leq \tau_{max}$ que se calculan de forma heurística. Inicializa todos los rastros de feromona al máximo valor τ_{max}, en lugar de a un valor pequeño τ_0. De esta forma, al aplicar la regla de actualización, los arcos de las buenas soluciones mantienen valores altos mientras que los de las malas los reducen. Además, da lugar a una mayor exploración al comienzo de la ejecución del algoritmo.

- *Reinicia la búsqueda* cuando se estanca, volviendo a poner todos los rastros de feromona a τ_{max}.

La combinación de todas estas reglas establece un buen balance *exploración-explotación* y reduce la posibilidad de estancamiento de la búsqueda, lo que convierte a este mecanismo en un algoritmo muy competitivo.

Una nueva extensión de los sistemas de hormigas, basada en la incorporación de componentes de *Computación Evolutiva* para mejorar el equilibrio *intensificación-diversificación*, es el *Sistema de Hormigas Mejor-Peor* [23, 22]. La nueva extensión mantiene la regla de transición del sistema de hormigas original y modifica los siguientes aspectos:

- El mecanismo de actualización de rastros de feromona evapora éstas de todos los caminos, reforzando positivamente sólo los de la mejor solución global y negativamente los de la peor solución actual. Es decir, además de la evaporación global, se aplica una evaporación adicional de los rastros de feromona de la peor solución de la iteración actual que no estén contenidos en la mejor solución global.

- Aplica una mutación de los rastros de feromona para diversificar. La mutación se aplica en cada rastro de feromona con cierta probabilidad, aumentando la *fuerza* de la mutación con las iteraciones.

- Reinicia la búsqueda cuando se estanca, como se hace en el *Sistema de Hormigas Max-Min*. Este sistema considera la búsqueda estancada cuando durante un número consecutivo de iteraciones no se consigue mejorar la mejor solución global obtenida. En este caso, se establecen todos los rastros de feromonas a τ_0.

Se debe tener en cuenta que es posible *hibridar* los algoritmos de hormigas con técnicas de *búsqueda local* para mejorar su eficacia. Esta *hibridación* consistiría en aplicar una búsqueda local sobre las soluciones construidas por todas las hormigas en cada iteración antes de

actualizar la feromona, es decir, se ejecutaría el algoritmo como si sólo existiesen las aristas que han sido recorridas por alguna de las hormigas que han encontrado una solución. El aumento de eficacia conlleva una disminución en la eficiencia, por lo que es habitual emplear la búsqueda local junto con las llamadas *listas de candidatos*, que consiste en estudiar sólo los pasos más prometedores en cada paso de la hormiga.

El *algoritmo básico* de sistemas de hormigas con búsqueda local es:

```
algoritmo HormigasConBusquedaLocal
   mientras no condición de parada hacer
      Construcción probabilística de las soluciones preliminares
         mediante la colonia de hormigas
      Refinamiento de dichas soluciones mediante búsqueda local
      Actualización global de la feromona
   fin mientras
fin algoritmo
```

De este modo, los algoritmos de hormigas con búsqueda local son algoritmos híbridos con una generación probabilística de soluciones iniciales basada en una técnica clásica de búsqueda local. Así, podría encuadrarse dentro de los algoritmos de *Búsqueda Local Multiarranque*, aunque éstos últimos generan aleatoriamente las soluciones iniciales sin utilizar información heurística. La principal diferencia es que el uso de las hormigas da lugar a un *mecanismo de cooperación global* entre las soluciones generadas, lo que hace que las ejecuciones no sean independientes entre sí.

A posteriori, cuando la mayoría de los algoritmos de hormigas estaban ya propuestos, *Dorigo y Di Caro* propusieron un marco de trabajo general que define la *metaheurística* de hormigas [32]. Instancian el algoritmo general con componentes concretas, como las reglas de transición, las actualizaciones de los rastros de feromonas, etc. pudiendo así obtener distintas variantes de algoritmos de hormigas. Los distintos algoritmos se pueden implementar de forma secuencial o paralela para ser aplicados respectivamente a problemas estáticos o dinámicos.

En general, para aplicar los algoritmos de hormigas a un problema, es necesario que el problema se pueda representar en forma de grafo con pesos. Cada arco del grafo contendrá dos tipos de información:

- *Información heurística*: que muestra la preferencia de las hormigas por escoger un arco en el camino y depende del caso concreto del problema. Esta información no es modificada por las hormigas durante la ejecución del algoritmo, aunque puede variar a lo largo del tiempo en problemas dinámicos.
- *Información memorística*: que es una medida de la *deseabilidad* del arco, representada por la cantidad de feromona depositada en él y modificada durante el algoritmo.

A modo de ejemplo, los algoritmos de optimización basados en colonias de hormigas se han aplicado al *enrutamiento* de paquetes en redes de telecomunicaciones [20, 93]. El *enrutamiento* es la tarea consistente en determinar el camino que seguirán los paquetes en una red de telecomunicaciones cuando llegan a un nodo, de forma que puedan alcanzar su nodo destino de la forma más rápida posible. Un ejemplo de este tipo de algoritmo es *AntNet*, que es un algoritmo de hormigas adaptativo y distribuido para enrutamiento de paquetes en redes.

Las redes se pueden modelar mediante un grafo dirigido con N nodos de procesamiento/destino, donde los arcos del grafo están caracterizados por el ancho de banda (en bits/segundo) y el retardo de transmisión (en segundos) del enlace físico. Se consideran dos tipos de paquetes: los de enrutamiento y los de datos, teniendo una mayor prioridad los paquetes de enrutamiento.

Las hormigas se consideran equivalentes a los paquetes de enrutamiento, lanzándose asíncronamente a la red hacia nodos destino aleatorios. Cada una de las hormigas buscará un camino de coste mínimo entre su nodo de partida y su nodo de destino. Para ello, se moverá paso a paso por la red (representada por el grafo) y en cada nodo intermedio tendrá que decidir a qué nodo se dirige mediante una regla de transición. Para tomar esta decisión, la hormiga debe considerar la cantidad de feromona y la preferencia heurística de los enlaces de la red.

Como es de esperar, el estado de la red varía con el tiempo por la posible caída de enlaces, la congestión, etc. Ahora bien, el algoritmo maneja adecuadamente esta dificultad gracias a su naturaleza distribuida y su capacidad de adaptación.

Cuando una hormiga llega a su nodo de destino, vuelve sobre sus pasos actualizando en el camino las tablas de enrutamiento de los nodos de acuerdo al tiempo que tardó en hacer el camino, reforzando las rutas positivamente o negativamente.

Este tipo de algoritmos consigue resolver de manera subóptima problemas NP-duros eficientemente, como por ejemplo el problema del *viajante de comercio*, el problema del

coloreado de grafos o la *secuenciación de tareas*, presentando ciertas ventajas y desventajas frente a otros métodos. Por ejemplo, en el problema del *viajante de comercio* es relativamente eficiente, pues ACO tiene un mejor comportamiento que otras técnicas de optimización global para el problema TSP, como por ejemplo que las *redes neuronales*, los algoritmos genéticos o el enfriamiento simulado. Comparado con los algoritmos genéticos, el método ACO retiene en memoria los datos de la colonia de hormigas entera, mientras que en los algoritmos genéticos sólo se mantiene la información de la anterior generación. Sin embargo, al método de las hormigas le afecta menos la elección de malas soluciones iniciales, gracias a la combinación de la elección aleatoria del camino y la memoria de las hormigas. Además, en los algoritmos ACO la convergencia está garantizada aunque el tiempo de convergencia se desconoce.

2.6. Algoritmos Basados en la Formación Dinámica de los Ríos

En esta sección se considera otro marco de trabajo. Este marco de trabajo alternativo es un diseño original de esta tesis y también está basado en la naturaleza, en particular en la *formación dinámica de los ríos*. Imaginemos que se deposita una masa de agua en algún punto a cierta altura. La gravedad hará que esa masa de agua siga algún camino hacia abajo hasta que no pueda bajar más. En términos de *Geología*, cuando llueve en las montañas - Figura 2.17 -, se depositan masas de agua en dichas montañas - Figura 2.18 - y el agua trata de encontrar su propio camino hacia abajo para llegar al mar. A lo largo del camino, el agua erosiona la tierra - Figura 2.19 - y transforma el paisaje creando un cauce. Cuando el agua atraviesa una fuerte pendiente, erosiona sedimentos de la tierra en su camino - Figura 2.20 -. Estos sedimentos serán depositados más tarde cuando la pendiente sea menor - Figura 2.21 -. Los ríos afectan al medio reduciendo (es decir, erosionando) o aumentando (es decir, sedimentando) la altitud del terreno - Figura 2.22 -.

Resaltemos que, si el agua se deposita en todos los puntos del terreno (por ejemplo, llueve), entonces la forma del río tiende a optimizar la tarea de recoger todo el agua y llevarla hasta el mar, lo que no implica coger el camino más corto hasta el mar desde un punto de origen *dado*. Destaquemos que hay *muchos* puntos de origen a considerar (uno por cada punto en el que cae una gota). De hecho, en este caso se creará una especie de *camino más corto combinado*. Sin embargo, si el agua fluye desde un punto *único* y no se consideran otras fuentes de agua, entonces el camino del agua tiende a seguir la forma más eficiente para reducir la altura (es decir, trata de encontrar el camino más corto).

A continuación se presenta el algoritmo basado en la formación dinámica de los ríos. El

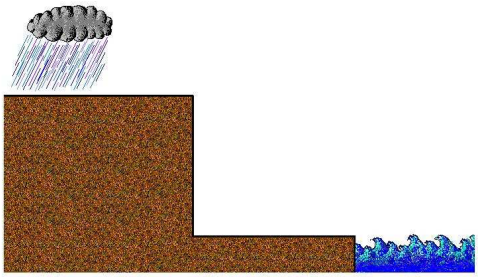

Figura 2.17: Llueve en las montañas

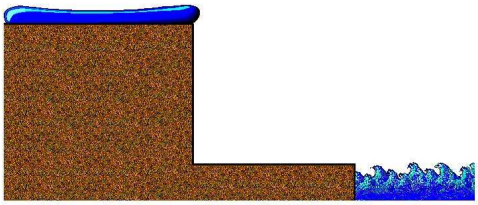

Figura 2.18: Se forma una masa de agua

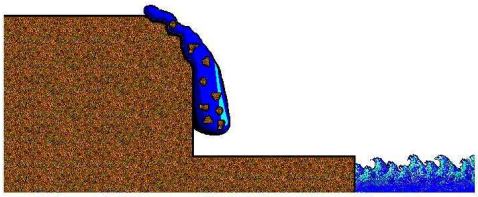

Figura 2.19: El agua avanza y produce erosión

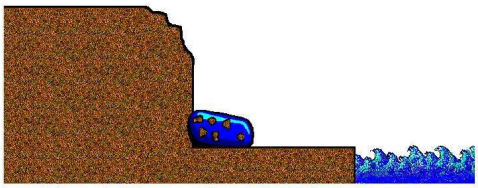

Figura 2.20: El agua arrastra sedimentos

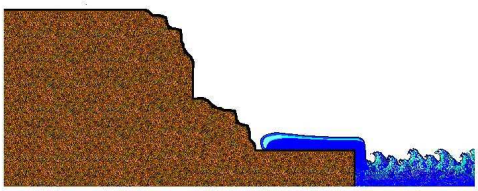

Figura 2.21: El agua alcanza el mar

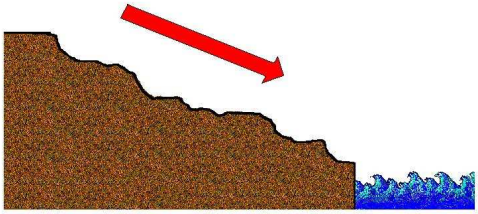

Figura 2.22: Se producen pendientes de bajada

método funciona de la siguiente manera. En vez de asociar valores de feromonas a las aristas como en los algoritmos de colonias de hormigas, se asocian valores de *altura* a los nodos del grafo. Cuando se mueven las gotas, o bien erosionan el terreno (reduciendo la altura de los nodos), o bien depositan el sedimento (aumentando la altura de los nodos). La probabilidad de que una gota elija una arista dada en lugar de otra es proporcional a la pendiente de bajada de la arista y, por tanto, depende de la diferencia de alturas entre los nodos y la distancia (es decir, el coste de la arista). Al comienzo del algoritmo, se parte de un mundo *plano*, esto es, todos los nodos tienen la misma altura. La excepción es el nodo destino, que es un *agujero*. Las gotas se depositan en el nodo de origen y se extienden por todo el terreno llano hasta que algunas de las gotas caen en el nodo destino. Este hecho erosionará a los nodos adyacentes, lo que creará nuevas pendientes de bajada y, de este modo, se propagará el proceso de erosión. Después se insertarán nuevas gotas en el nodo origen para transformar los caminos y reforzar la erosión de los caminos prometedores. Después de algunos pasos, se encontrarán buenos caminos desde el origen hasta el destino. Estos caminos vienen dados en forma de secuencias de aristas decrecientes desde el origen hasta el destino.

Este nuevo método, que denominamos RFD (del inglés River Formation Dynamics), proporciona las siguientes ventajas respecto al ACO. Por un lado, los ciclos locales no se crearán ni reforzarán porque esto implicaría un *ciclo siempre decreciente*, lo que es contradictorio. Si bien las hormigas tienen en cuenta el camino ya realizado para poder evitar repetir nodos, no pueden evitar ser conducidas por rastros de feromonas a través de algunas aristas de tal manera que sea inevitable repetir algún nodo en el siguiente paso[1]. Por el contrario, las *alturas* no pueden guiar a las gotas a estas situaciones. Por otro lado, cuando RFD encuentra un camino más corto, el posterior refuerzo del camino es rápido: si se tienen el mismo origen y el mismo destino tanto en el camino nuevo como en el camino viejo, entonces la diferencia de altura es obviamente la misma pero la distancia es diferente. Por lo tanto, las aristas del camino más corto tienen necesariamente mayores pendientes de bajada (en media) y son preferidas inmediatamente (en media) por las gotas que vengan a continuación. Además, nótese que el proceso de erosión proporciona un método para evitar soluciones ineficientes: si un camino conduce a un nodo que está a menor altura que el resto de nodos adyacentes (es decir, es un *callejón sin salida*) entonces la gota depositará su sedimento, incrementando la altura del nodo. Antes o después, la altura de este nodo alcanzará la altura de sus vecinos, con lo que se evitará que otras gotas queden atrapadas en este nodo. Es más, antes de llegar a ese punto las gotas podrían acumularse en el nodo hasta que la masa de agua alcanzase la

[1]Normalmente, este hecho implica repetir el nodo o *matar* a la hormiga. En ambos casos, los últimos movimientos de la hormiga fueron inútiles.

altura de los nodos vecinos (se forma un *lago*). Si el agua alcanzara este nivel, otras gotas serían capaces de cruzar este nodo desde un nodo adyacente a otro. De esta manera, los caminos quedarán interrumpidos hasta que los sedimentos no rellenen el hueco. Esta técnica proporciona un método implícito para evitar comportamientos ineficientes de las gotas.

2.6.1. Algoritmo Básico

El esquema básico del algoritmo es el siguiente:

```
inicializarGotas()
inicializarNodos()
mientras (no todasGotasSiguenMismoCamino()) y (no otraCondicionFin())
   moverGotas()
   erosionarCaminos()
   depositarSedimentos()
   analizarCaminos()
fin mientras
```

Este esquema muestra la idea principal del algoritmo propuesto. A continuación se analiza el comportamiento de cada uno de los pasos. Primero, se inicializan las gotas (`inicializarGotas()`), es decir, se sitúan todas las gotas en el nodo inicial. El siguiente paso consiste en inicializar todos los nodos del grafo (`inicializarNodos()`). En este paso se realizan dos operaciones. En la primera operación, se fija la altura del nodo destino a 0. En términos de la analogía de la formación dinámica de los ríos, este nodo representa el *mar*, es decir, el objetivo final de todas las gotas. En la segunda operación, se asigna el mismo valor para la altura del resto de los nodos.

El bucle `mientras` del algoritmo se ejecuta hasta que todas las gotas encuentran la misma solución (`todasGotasSiguenMismoCamino()`), es decir, todas las gotas atraviesan la misma secuencia de nodos, o se satisface otra condición de finalización (`otraCondicionFin()`). Esta condición se puede utilizar para, por ejemplo, limitar el número de iteraciones o el tiempo de ejecución. Otra posibilidad es la de finalizar el bucle si la mejor solución encontrada hasta el momento no ha sido mejorada en las últimas N iteraciones.

El primer paso del cuerpo del bucle consiste en ir moviendo las gotas por los nodos del grafo (`moverGotas()`) de manera parcialmente aleatoria. En la siguiente *regla de transición* se define la probabilidad de que una gota k que se encuentra en el nodo i elija al nodo j como destino:

$$P_k(i,j) = \begin{cases} \frac{pendienteBajada(i,j)}{\sum_{l \in V_k(i)} pendienteBajada(i,l)} & \texttt{if} \quad j \in V_k(i) \\ 0 & \texttt{if} \quad j \notin V_k(i) \end{cases} \qquad (1)$$

donde $V_k(i)$ es el conjunto de nodos *vecinos* del nodo i que pueden ser visitados por la gota k y *pendienteBajada(i,j)* representa la pendiente entre los nodos i y j, y se define del siguiente modo:

$$pendienteBajada(i,j) = \frac{altura(j) - altura(i)}{distancia(i,j)} \qquad (2)$$

donde *altura(x)* es la altura del nodo x y *distancia(i,j)* es la longitud de la arista que conecta los nodos i y j. Fijémonos en que, al principio del algoritmo, la altura de todos los nodos es la misma, por lo que $\forall i \sum_{l \in V_k(i)} pendienteBajada(i,l)$ es 0. Para dar un tratamiento especial a las pendientes planas, se modifica este esquema como sigue: la probabilidad de que una gota se mueva a través de una arista con pendiente 0 se establece a un valor no nulo. Este tratamiento va a permitir que las gotas se dispersen en un medio plano, que es totalmente necesario, en particular al principio del algoritmo.

En la siguiente fase (`erosionarCaminos()`) se erosionan los caminos en consonancia con los movimientos de las gotas en la fase previa. En particular, si una gota se mueve de un nodo A a un nodo B, se erosiona el nodo A. Es decir, la altura de este nodo se reduce en función de la pendiente actual entre A y B. En particular, la erosión será mayor si la pendiente de bajada entre A y B es elevada. Si la arista es plana entonces se realiza una leve erosión. La altura del nodo final (es decir, el mar) nunca se modifica y permanece igual a 0 durante toda la ejecución.

Como se comentó anteriormente, el proceso de erosión evita en la práctica que las gotas sigan ciclos porque un ciclo debe incluir, al menos, una pendiente de subida y, de acuerdo al comportamiento básico explicado más arriba, las gotas no pueden *ascender*. Por el contrario, en ACO los rastros de feromonas presentes en caminos independientes pueden interferir entre sí de tal manera que las hormigas realimenten los ciclos y los sigan[2].

Además, supongamos que se encuentra un nuevo camino que es mejor que otros caminos considerados hasta el momento. En RFD, las diferencias de alturas favorecen rápidamente esta nueva ruta contra otras posibilidades. Esto es debido a que las pendientes de bajada del nuevo camino son preferibles *en media* a los otros viejos caminos: globalmente, en los

[2]Cabe destacar que, para alimentar un ciclo, no es necesario que una hormiga siga ese ciclo. Es suficiente con que cada hormiga refuerce una parte *diferente* del ciclo, de tal manera que todas las partes son reforzadas de manera individual.

caminos se atraviesan diferentes distancias acumuladas, pero la diferencia de alturas es la misma. Esto facilita la tarea siguiente de realimentar el nuevo camino de modo que los pasos no sólo sean preferidos en media, sino que cada uno de ellos sea preferido individualmente a las otras posibilidades. Por otro lado, cuando una hormiga encuentra un camino mejor por vez primera, el rastro de feromona es todavía insignificante comparado con los otros caminos antiguos. Por ello, este camino debe ser realimentado gradualmente hasta que las aristas del nuevo camino sean preferidas al menos *en media*.

Una vez que finaliza el proceso de erosión, se aumenta ligeramente la altura de los nodos del grafo (`depositarSedimentos()`). El objetivo es evitar que, después de varias iteraciones, el proceso de erosión lleve a una situación donde todas las alturas sean cercanas a 0, lo que haría que las pendientes fuesen despreciables y arruinaría todos los caminos formados.

Finalmente, el último paso (`analizarCaminos()`) realiza un estudio de todas las soluciones encontradas por las gotas y almacena la mejor solución hallada hasta el momento.

2.6.2. Mejoras Básicas

En esta sección se muestran algunas mejoras aplicadas al esquema básico. Como se anunció anteriormente, permitimos que las gotas se muevan por aristas con un gradiente igual a 0. Yendo un paso más lejos, se aplica la siguiente mejora: permitimos que las gotas asciendan por pendientes *crecientes* con una probabilidad baja. Esta probabilidad será inversamente proporcional a la pendiente de subida. Esta nueva característica mejora la búsqueda de buenos caminos. Nótese que las soluciones encontradas durante los primeros pasos del algoritmo predisponen la exploración del grafo en lo sucesivo. Esto es debido a que las gotas tienden a seguir los caminos formados previamente. Permitir a las gotas escalar pendientes crecientes con cierta probabilidad permite explorar y encontrar otros caminos alternativos en el grafo. Esta característica hace al método menos dependiente del comportamiento inicial en los primeros pasos. De hecho, la probabilidad de escalar pendientes crecientes encapsula la mayor parte de la dependencia del método en las soluciones previas. Como es natural en los algoritmos de búsquedas heurísticas, esta dependencia debe proporcionar un equilibrio adecuado entre las soluciones anteriores y las nuevas alternativas.

Así mismo, la probabilidad de escalar pendientes ascendientes se irá reduciendo a lo largo de la ejecución del algoritmo, esto es, para cada nueva iteración esta probabilidad se reduce ligeramente. Tras realizar algunas iteraciones, la exploración del grafo será suficiente, por lo que se reducirá la necesidad de buscar más caminos alternativos (recordemos que se reduce la probabilidad de escalar pendientes, pero la probabilidad de escoger aristas que no tienen

la máxima pendiente decreciente permanece igual). Además, adoptamos la siguiente idea del *Enfriamiento Simulado* (véase la sección 2.2). Además de reducir constantemente la probabilidad de que una gota tome una pendiente creciente, añadimos la siguiente característica: cada N iteraciones, esta probabilidad se aumentará en vez de reducirse ligeramente. Por un lado, esto reduce la dependencia de (malas) viejas soluciones bien establecidas, pues así se permite buscar otras rutas alternativas periódicamente. Por otro lado, los caminos que son realmente buenos *sobrevivirán* a este *desajuste* periódico. Préstese atención a que los caminos malos a corto plazo pueden volverse buenos a largo plazo. Es por ello que esta técnica ayuda a evitar quedarse bloqueado en un óptimo local. La probabilidad de escoger pendientes crecientes se incrementa de tal modo que, globalmente, el decremento lo supera, es decir, dicha probabilidad tiende a ser 0.

Permitir que las gotas asciendan pendientes implica que la probabilidad de que una gota realice un ciclo ya no es 0. Sin embargo, nuestro método todavía provee una manera eficiente para evitar que las gotas atraviesen un ciclo en la práctica. Por un lado, esta probabilidad es baja porque un ciclo debe contener, al menos, una pendiente creciente. En particular, si sólo hay una pendiente creciente entonces esta debe compensar el resto de pendientes decrecientes. Por tanto, esa pendiente creciente debe ser elevada. Dado que la probabilidad de escalar una pendiente creciente es inversamente proporcional a la magnitud de la pendiente, la probabilidad de escalar esta pendiente es muy baja. Además, como se dijo anteriormente, la probabilidad de ascender pendientes se reduce a lo largo del tiempo. Por otro lado, la probabilidad de que una hormiga siga un ciclo local en ACO *aumenta* mientras sigan siendo reforzadas todas las aristas del ciclo (recordemos que cada arista podría ser reforzada por hormigas que sigan caminos diferentes).

Otra característica que se esbozó anteriormente, pero que no se consideró en el esquema básico presentado anteriormente, consiste en permitir a las gotas *depositar sedimentos* en los nodos. Esto sucede cuando todos los movimientos posibles para una gota implican trepar una pendiente creciente y la gota no consigue escalar ninguna de las aristas (de acuerdo a la probabilidad asignada para ello). En este caso, la gota se queda bloqueada y deposita el sedimento que transporta, aumentando la altura del nodo actual. Este incremento es proporcional a la cantidad de sedimento acumulada. Como se comentó anteriormente, esta sedimentación va a permitir *penalizar* gradualmente a los caminos que conducen a callejones sin salida. Tarde o temprano, el sedimento acumulado incrementará la altura de este nodo hasta que alcance la altura de los nodos cercanos. En ese mismo instante, las pendientes que conduzcan a este nodo ya no serán decrecientes. Este mecanismo previene que otras gotas

sigan este camino.

La última mejora consiste en *agrupar* las gotas para reducir el número de movimientos individuales. Las gotas que están en el mismo nodo se agruparán formando una única gota de mayor tamaño (es decir, se incrementa el caudal). De este modo, se puede manejar una única gota de tamaño N en vez de N gotas de tamaño 1. Al comienzo, se pone una gota de tamaño N en el primer nodo. A continuación, cuando se mueve la gota y existen varias alternativas de movimiento, la gota se divide en gotas de menor tamaño, y cada una de ellas sigue un camino diferente. Agrupar las gotas permite reducir el número de decisiones realizadas por el algoritmo en cada paso porque hay menos gotas que considerar: en vez de gastar esfuerzo computacional en mover cada gota, se toma una sola decisión para cada grupo, consiguiendo mejorar la eficiencia del método. Resaltemos que, en el caso peor, una gota se divide en otras de tamaño 1 y cada una ya no se une a ningún caudal alternativo más tarde, es decir, el tamaño de cada una permanece igual a 1. Así, se estaría en el caso donde las gotas no están agrupadas.

Cuando se mueve una gota *grande* se actúa como sigue. Supongamos que $trunc(X)$ devuelve la parte entera de X. Primero, se tienen en cuenta las probabilidades de elegir cada alternativa como si el tamaño de la gota fuese 1. Luego, se usan estas probabilidades para dividir la gota *de forma determinista*, moviendo cada una a su destino. Se realiza de la siguiente manera: si hay una gota de tamaño 204 en el nodo A y es posible moverse a B, C o D, con probabilidades 0,35, 0,22 y 0,43 respectivamente, entonces una gota de tamaño $\texttt{trunc}(204 \cdot 0{,}35) = \texttt{trunc}(71{,}40) = 71$ se mueve a A, una gota de tamaño $\texttt{trunc}(204 \cdot 0{,}22) = \texttt{trunc}(44{,}88) = 44$ se mueve a C y una gota de tamaño $\texttt{trunc}(204 \cdot 0{,}43) = \texttt{trunc}(87{,}72) = 87$ se mueve a D. Las restantes $204 - (71 + 44 + 87) = 2$ gotas se mueven (aleatoriamente) como gotas simples aplicando la fórmula (1) presentada en la sección 2.6.1.

Capítulo 3

Métodos Evolutivos y Métodos Formales

La comunidad de *Métodos Formales* [15, 28, 46, 94, 95] se ha dedicado desde hace décadas a diseñar métodos matemáticos para analizar la corrección de los sistemas. Un método formal se puede definir como cualquier técnica que trate la construcción y/o el análisis de modelos matemáticos que contribuyen a la automatización del desarrollo de sistemas informáticos (tanto hardware como software) [2]. Con estos métodos formales basados en el empleo de técnicas, lenguajes y herramientas definidas matemáticamente se desea cumplir objetivos tales como facilitar el análisis y construcción de sistemas fiables independientemente de su complejidad, poniendo de manifiesto posibles inconsistencias, ambigüedades o errores que podrían pasar inadvertidos. Hay dos partes principales en los métodos formales: la especificación formal (donde se usan las matemáticas para especificar las propiedades deseadas de un sistema) y la verificación formal (en la que se usan las matemáticas para probar que un sistema satisface una especificación). A lo que quizás se podría añadir: la semi-automatización del proceso de generación de programas.

En los últimos años, la idea de que la formalización matemática del software es el enfoque más apropiado para conseguir mejorar su calidad va adquiriendo cada vez más fuerza. Los partidarios de los métodos formales defienden que su empleo, a lo largo de todo el ciclo de vida, facilita el desarrollo de especificaciones claras, concisas y no ambiguas, permitiendo el análisis funcional de la especificación y posibilitando el desarrollo de implementaciones correctas respecto a su especificación. Sin embargo, los detractores aseguran que el empleo de métodos formales supone un volumen de trabajo considerable, aumento en los costes y tiempo de desarrollo y que debe quedar supeditado a herramientas que lo automaticen.

Sin embargo, no se pueden poner en duda las ventajas de las técnicas formales. Estas técnicas aportan una *mejor comprensión* del sistema; una *mejora en la comunicación con el cliente*, ya que se dispone de una descripción clara y no ambigua de los requisitos, eliminando uno de los problemas clave de las especificaciones informales y las descripciones ambiguas escritas como texto; una *mayor precisión* en la descripción del sistema, ya que con las matemáticas se pueden expresar propiedades complejas de forma concisa; una *mayor calidad software* respecto al cumplimiento de las especificaciones; y *mayor productividad*. El uso de las matemáticas facilita las demostraciones y las especificaciones formales para que éstas puedan manipularse usando ordenadores. De este modo y usando técnicas matemáticas, será posible hacer software *más correcto*.

Por otro lado, la posible falta de madurez en la práctica de los métodos formales causa problemas a la hora de su utilización a nivel industrial. Algunas de estas causas son las siguientes: el desarrollo de herramientas que apoyan la aplicación de métodos formales es a veces complicado; la colaboración entre la industria y el mundo académico es inferior a lo que sería deseable; hay cierta creencia que la aplicación de métodos formales encarece los productos y prolonga su desarrollo.

A pesar de estos problemas, los métodos formales se han aplicado de manera exitosa a muchos problemas industriales pero se encuentran típicamente con un problema en la práctica: el número de estados a analizar sistemáticamente crece exponencialmente con el tamaño del sistema a validar. Es por ello que las técnicas exhaustivas para encontrar errores en el sistema son sustituidas por estrategias heurísticas que permiten focalizar la búsqueda de errores potenciales en configuraciones sospechosas o críticas. Recientemente, algunos grupos de investigación de métodos formales han reconocido el potencial de los métodos de *Computación Evolutiva* para utilizarlos como heurísticas en estos problemas. De hecho, los métodos de computación evolutiva proporcionan eficientes estrategias genéricas para encontrar buenas soluciones en espacios de soluciones muy grandes, lo cual se adecúa perfectamente a los problemas típicos que aparecen en los métodos formales.

Así como en el capítulo anterior fueron presentados los principales métodos evolutivos, en las siguientes secciones hablaremos de dos métodos formales, el *testing formal* y el *model checking*, y de cómo se pueden aplicar algunos de los métodos evolutivos estudiados anteriormente a la resolución de problemas que aparecen en los métodos formales citados.

3.1. Testing Formal

La investigación en métodos formales ha llevado a muchos lenguajes formales y técnicas de verificación (apoyadas por prototipos de herramientas) a verificar propiedades de *alto nivel* en descripciones formales de sistemas. Aunque estos métodos se basan en importantes teorías matemáticas, en nuestros tiempos no hay muchos sistemas desarrollados cuya corrección esté completamente verificada formalmente. Esto es debido a que la práctica actual para verificar sistemas se basa en un enfoque más práctico e informal. El *testing* [101, 102, 9, 64, 47, 67, 91, 90] es normalmente la técnica predominante, donde una implementación tiene que pasar una serie de tests que han sido obtenidos de manera heurística o diseñados específicamente para ese fin, observando el comportamiento de los tests durante su ejecución y asignando un veredicto sobre el correcto funcionamiento de la implementación. El criterio de corrección a testear se debe definir en la especificación del sistema. La especificación describe lo que el sistema tiene que hacer y lo que no, por lo que constituye la base de cualquier actividad de testing. La combinación del testing con los métodos formales no ha sido tradicionalmente muy habitual, ya que se hace difícil imaginar cómo la parte práctica del testing se puede combinar con la parte matemática de la verificación usando métodos formales.

El proceso de *testear* un sistema software es una tarea importante que resulta costosa y larga donde se consume entre el 30 % y el 50 % del coste total del desarrollo del software [36]. La dinámica general es que está aumentando el esfuerzo empleado en el testing debido a la búsqueda continua de software de mejor calidad y el crecimiento de los sistemas en tamaño y complejidad. La situación es más problemática porque la complejidad del testing tiende a aumentar más rápidamente que la complejidad del sistema a testear y en el peor de los casos crece exponencialmente. Muchos de estos problemas son atribuidos al testing equivocadamente ya que muchos de ellos surgen porque la especificación del sistema no es ni clara, ni precisa, ni completa y, además, es ambigua. Si se parte de una mala especificación para el proceso de testing, entonces acarreará problemas y dificultades de interpretación y necesitará clarificaciones de las intenciones de la especificación. Este hecho puede conllevar a tener que rehacer la especificación de nuevo durante la fase de testing del desarrollo software.

La mejora del proceso de testing de software se ha orientado al desarrollo de técnicas que soporten su automatización, ya que se traducirá en un gran ahorro en costes. La automatización de esta tarea parece una solución lógica. Por un lado ayudará a realizar el proceso del testing más rápidamente y también lo hará menos susceptible a errores humanos e independiente a interpretaciones humanas. El *testing sistemático* es una de las técnicas más importantes y ampliamente usadas para comprobar la calidad del software. Existen muchas

técnicas de testing que tratan de aumentar la calidad del producto final mediante el incremento de su nivel de *efectividad* y *eficiencia*. La efectividad se optimiza mediante el uso de conjuntos de tests que proporcionen una alta probabilidad de detectar fallos, mientras que la eficiencia se incrementa reduciendo el número de tests que deben ser ejecutados para alcanzar dicho objetivo. La mayoría de las herramientas de automatización soportan la ejecución de procesos de tests. Esto incluye la ejecución, el almacenamiento automático y la re-ejecución de casos específicos de tests. A menudo, los casos de test son escritos manualmente en un lenguaje especial de *scripts* específico de cada herramienta. Una vez escritos, estos scripts se pueden ejecutar automáticamente.

Las herramientas de ejecución de tests no suelen ayudar en el desarrollo de los casos de test. Habitualmente, los casos de test tienen que ser desarrollados por humanos que, después de leer y estudiar la especificación, piensan sobre qué se tiene que probar y sobre cómo escribir esos scripts para que ejecuten aquello que desean probar. Además no existen muchas herramientas disponibles que nos puedan ayudar con el proceso de la generación de buenos tests a partir de la especificación. Precisamente, uno de los principales cuellos de botella para automatizar el proceso de generación de tests es la calidad de las especificaciones. Por un lado, son imprecisas, están incompletas y son ambiguas como ya se dijo anteriormente y, por otro lado, las especificaciones actuales suelen estar escritas en lenguaje natural (inglés, español, etc.) y no son fácilmente combinables con las herramientas para la derivación sistemática de casos de test.

Una de las tecnologías más prometedoras para resolver el desafío del testing de software es el *model-based testing* (o testing basado en modelos) que es testing de software [11, 35, 103] donde los tests se obtienen de un modelo que describe algunas características (normalmente funcionales) del *sistema bajo test* (normalmente abreviado como SUT, del inglés *system under test*). En el *model-based testing*, se testea un SUT frente a una descripción formal de su comportamiento llamada *modelo*. Este modelo se usa como una especificación completa y precisa de lo que debería hacer el SUT y es una base adecuada para realizar el testing. Una de las ventajas de estos modelos es que se pueden procesar automáticamente con herramientas que hacen posible la automatización del proceso del testing. Este hecho conduce a que la generación de tests se realice de una manera más rápida y con menos errores, ya que se pueden automatizar la generación de millones de tests a partir del modelo y, finalmente, ejecutarlos y analizarlos. Si el modelo es válido, es decir, si expresa exactamente lo que el SUT debería hacer, entonces todos estos tests son, probablemente, también válidos.

3.2. Model Checking

En los últimos 10 ó 20 años, la comunidad investigadora en informática ha realizado tremendos progresos en el desarrollo de herramientas y técnicas para verificar requisitos y diseño. Una de las propuestas más prometedoras que ha emergido es el llamado *model checking* [19, 106, 68, 73]. Dado un modelo abstracto y simplificado de un programa, el model checking consiste en comprobar automáticamente si el modelo satisface una especificación también dada. Cuando se combina con el uso estricto de un lenguaje formal para modelar, se puede automatizar el proceso de verificación con cierta facilidad. La idea es la siguiente: una herramienta de model checking acepta los requisitos del sistema o el diseño (llamados *modelos*) y una propiedad (llamada *especificación*) que se espera que se satisfaga en el sistema final. Si el modelo dado cumple las especificaciones dadas, la herramienta entonces devuelve como salida un *sí* y, en caso contrario, crea un contraejemplo. De esta manera, cada ejecución de la herramienta puede proveer una valiosa retroalimentación. La idea consiste en que asegurando que el modelo satisface suficientes propiedades del sistema, se incrementará la confianza en la corrección del modelo. Cabe destacar que en la práctica esta visión es a veces idealizada ya que, a menudo, los recursos disponibles sólo permiten analizar una parte del modelo del sistema que deseamos analizar. Por eso es muy importante que el modelo abstracto refleje adecuadamente el comportamiento concreto del sistema. Concluimos que el model checking no es un sustituto de los procedimientos estándar para asegurar la calidad de un sistema, sino que es una técnica adicional que puede ayudar a descubrir problemas de diseño en los primeros pasos del desarrollo software. Algunos aspectos que diferencian al model checking de la verificación tradicional es que es totalmente algorítmico, de baja complejidad computacional, no requiere intervención alguna del verificador y, además, devuelve información cuando no puede verificar una propiedad.

Cuando comprobamos requisitos del sistema, básicamente se intenta encontrar la respuesta a algunas preguntas: ¿reflejan con precisión los requisitos del usuario?, ¿los requisitos están escritos sin ambigüedades y de manera clara?, ¿son flexibles y pueden ser implementados por los ingenieros?, ¿se pueden usar los requisitos para definir de manera sencilla tests de aceptación para chequear si la implementación los cumple?, ¿los requisitos están escritos de manera abstracta y en alto nivel?

Encontrar las respuestas a estas preguntas es una gran tarea y no existe una manera sencilla para hacerlo. Se puede encontrar algo de ayuda en herramientas de modelado como UML [4], pero permanece el problema de asegurar la calidad de los requisitos. Este proceso es casi totalmente manual y conlleva un importante gasto en tiempo. Además, el coste de los

errores en el diseño de los requisitos es alto frecuentemente. Si se implementan unos requisitos incorrectos, conducirá a comportamientos inadecuados del sistema y costes elevados.

Un modo de mejorar la calidad de nuestros requisitos y diseño es usar herramientas automáticas para comprobar la calidad de varias de estas características. Es necesario implementar un lenguaje formal para el diseño de requisitos que sea claro, riguroso y que no sea ambiguo. Si el lenguaje para escribir estos requisitos y el diseño tienen bien definida la semántica, será posible desarrollar herramientas para analizar las sentencias escritas en ese lenguaje.

En sistemas orientadas a control, se aceptan las máquinas de estados finitos (FSM, del inglés Finite State Machine) como una notación buena, clara y abstracta para definir los requisitos del diseño de un sistema y serán la entrada del model checker junto a las propiedades que se desean comprobar, que habitualmente se expresan como fórmulas de lógica temporal. Sin embargo, una FSM *pura* no es adecuada para modelar sistemas industriales complejos en la vida real. Es por ello que es necesario emplear máquinas de estados finitos extendidas (EFSM, del inglés Extended FSM) para el modelado. La mayoría de las herramientas de model checking poseen su propio lenguaje formal para definir los modelos, pero la mayoría de ellas son alguna variante de las EFSM.

Se ha comprobado que el model checking es una tecnología muy exitosa para comprobar los requisitos y el diseño de una amplia variedad de sistemas, particularmente en sistemas hardware, sistemas empotrados en tiempo real y en sistemas de seguridad críticos. En cambio, inicialmente fueron aplicados a sistemas que se dedican más al control que a los datos, los llamados *sistemas reactivos* [78], que son aquellos sistemas cuyo rol consiste más en interaccionar con el ambiente que en aplicar transformaciones a datos complejos.

Una vez expuestas las ventajas de emplear model checking, veamos ahora cuáles son los principales inconvenientes cuando se emplea esta técnica en la práctica:

- Toda herramienta de model checking tiene su propio lenguaje de modelado, por lo que no es posible traducir automáticamente descripciones de requisitos informales a este lenguaje.

- Existen problemas similares para la notación de la especificación de propiedades, pues también es un lenguaje específico (a menudo una variante de *Computation Tree Logic*, CTL, o de *Linear Temporal Logic*, LTL). Algunas propiedades que se tengan que verificar serán difíciles o incluso imposibles de expresar empleando la notación escogida.

- Otra limitación cuando el model checking se aplica a sistemas reales es el problema

de la explosión de estados: el número de estados existentes en el espacio de estados de sistemas grandes y complejos puede ser enorme, incluso infinito, por lo que en la práctica no es posible realizar una búsqueda exhaustiva en el espacio de estados. No obstante, existen técnicas eficaces para trabajar con este problema, en las que sólo se construye una parte del espacio de estados del programa o en las que se representan implícitamente (en vez de explícitamente) los estados del programa.

3.3. Estudio de Aplicaciones de Métodos de Computación Evolutiva a los Métodos Formales

La aplicación de técnicas de Inteligencia Artificial (IA) a la Ingeniería del Software (IS) es un área de desarrollo emergente que mezcla ideas de dos dominios diferentes. Varias publicaciones (véase por ejemplo [10, 77, 17, 88]) han comenzado a examinar el uso efectivo de la IA en actividades relacionadas con la IS que inherentemente están centradas en el conocimiento humano. En particular, se ha identificado al testing de software como una de las áreas de la IS que más prolíficamente usan técnicas de IA. Estas técnicas implican la aplicación de *Algoritmos Genéticos* (AGs) [66, 61], el *Enfriamiento Simulado* [100], la *Escalada* [43], los *Algoritmos basados en Colonias de Hormigas* [29, 62] o la *Optimización basada en Nubes de Partículas* [49] para la generación de datos de tests.

Normalmente existen tres tareas principales asociadas con el testing de software: (1) la generación de datos de tests; (2) la ejecución de los tests implicando el uso de los datos de tests y el sistema analizado (SUT); (3) la evaluación de los resultados de los tests. Muchas de las herramientas para el *testeo* de modelos encontradas en la literatura usan algoritmos deterministas exactos para comprobar las propiedades. Estos algoritmos normalmente necesitan una cantidad enorme de recursos computacionales cuando el modelo testeado es grande, motivo por el cual algunos trabajos han considerado el empleo de métodos heurísticos basados en la computación evolutiva. Por otro lado, la idea de explotar tales heurísticas en el model checking ha recibido poca atención [40, 6, 7]. Esto es debido principalmente a dos razones. Primero, porque el model checking no es un problema de optimización: el objetivo principal no es encontrar la mejor solución (por ejemplo, el camino más corto que lleva a un estado), sino encontrar cualquier solución (por ejemplo, cualquier estado de error). Y segundo, en el model checking históricamente se le ha prestado mucha atención a la completitud donde el objetivo principal es comprobar de manera exhaustiva todos los estados alcanzables del sistema.

A continuación comentaremos algunos artículos importantes que aplican técnicas de computación evolutiva para resolver problemas de testing de software y de model checking.

En [13] se estudian estrategias que combinan técnicas de generación automática de juegos de tests con testing de secuencias largas o mucho volumen empleando un algoritmo genético. El testing de secuencias largas repite juegos de tests muchas veces, simulando intervalos de ejecuciones prolongadas. Estas técnicas de testing son prácticas para encontrar errores que surgen como problemas de coordinación de algunas componentes, así como de consumo de recursos del sistema (por ejemplo, falta de memoria) o datos corruptos. Asociar la generación automática de juegos de pruebas con el testing de secuencias largas hace que esta propuesta sea más escalable y efectiva.

Las técnicas de testing de software a menudo suponen desarrollar juegos de tests que consigan algún nivel de cobertura respecto al código objetivo. La meta es asegurar que todos (o al menos un subconjunto importante) de los *caminos* de ejecución se ejecuten. El testing de secuencias largas ha recibido menos atención. Consiste en que los sistemas se ejecutan durante largos periodos de tiempo repitiendo la ejecución de juegos de tests. A primera vista, los programadores esperarían que un test se ejecute de la misma manera en cualquier momento. No obstante, un test que se ha ejecutado satisfactoriamente repetidas veces puede repentinamente fallar después de prolongados periodos de ejecución. Los recursos del sistema se pueden consumir o volverse corruptos, y los relojes internos, contadores, así como otros componentes basados en el estado, pueden tomar valores incorrectos o simplemente desbordarse. Para motivar estos problemas, se describen algunos errores interesantes en sistemas distribuidos complejos. Todos estos errores ocurrieron después de que los sistemas estuvieran desarrollados. Un ejemplo de un error de ejecución ocurrió en el sistema de misiles *Patriot* durante la Guerra del Golfo de 1991. El sistema de misiles Patriot tenía la intención de interceptar y destruir los misiles *Scud*, pero un error en el software del procesamiento de los relojes internos hizo que fallara. Se suponía que estos relojes se reiniciaban frecuentemente, pero una ejecución prolongada durante más de 100 horas provocó que los misiles Patriot perdieran sus objetivos Scud. También se describen otros ejemplos de errores de sistema en el servicio de ambulancias de Londres o en el cambio de vía del ferrocarril en Alemania.

Las medidas tradicionales de cubrimiento de código no se ajustan a la idea de que los casos de test pueden empezar a fallar después de intentarlo repetidas veces debido al cambio de estado en sistemas complejos. Los algoritmos genéticos ofrecen una técnica con muchas

características deseables para el testing de secuencias largas demostrada por investigaciones anteriores (véase [75, 69, 70]). Por ejemplo, las técnicas automáticas pueden generar gran cantidad de casos de test sin la desviación inherente creada en la generación de casos de test hechos *a mano*. En particular, los algoritmos genéticos ofrecen una potente técnica de búsqueda que es efectiva en espacios de búsqueda muy grandes representados por atributos del sistema y parámetros de entrada en el testing. La función de fitness puede guiar una búsqueda que produce un gran número de juegos de tests localizados, pero también puede simular comportamientos más aleatorios dependiendo del escenario de testing con el que se esté trabajando.

En [13] se utiliza un algoritmo genético para generar buenos casos de prueba. Sin embargo, la noción de bondad depende de los resultados de los ciclos anteriores de testing. Es decir, se usa una función de bondad relativa en vez de absoluta. Para la creación de casos de prueba, la función de bondad relativa cambia a lo largo del tiempo con la población, permitiendo que las siguientes generaciones tomen ventaja de cualquier detalle recogido de los anteriores resultados almacenados en el *registro fósil*. Este registro fósil contiene todos los casos de test generados con anterioridad, información sobre el tipo de error generado (si es que lo hubo) y cuándo fue ejecutado dicho test.

El comportamiento de la búsqueda utilizada se puede describir usando un método basado en *exploradores, buscadores y mineros*. Un *explorador* es un juego de tests muy nuevo que se escoge independientemente de su aparente probabilidad de éxito. Estos exploradores son, esencialmente, casos de tests que se distribuyen a lo largo de toda la región del espacio de tests. Una vez que se descubre un error, la función de fitness fomenta que se testee más la región en la que se encontró el error. Los *buscadores* son casos de test únicos, pero que se encuentran cerca de los nuevos errores encontrados. De este modo, los tests nuevos y los próximos se combinan en la función de fitness para generar más buscadores. Los *mineros* se caracterizan por ser casos de test generados para explorar más profundamente aquellas áreas donde se encontraron errores en el pasado. De este modo se facilita la búsqueda de errores en nuevas zonas que no fueron investigadas ni por los exploradores ni por los buscadores. La mezcla de estos tres tipos de casos de test tiende a centrar la atención en zonas de errores mientras que también se generan algunos tests en regiones dispersas del espacio de tests.

Los experimentos realizados en el artículo muestran que el algoritmo genético desarrollado es capaz de generar datos de test que descubren fallos en el software con una precisión del 90 %-97 %, quedando demostrado el potencial de estas técnicas. Para finalizar se presenta el desarrollo de una aplicación para simular vehículos autónomos en la que se explora el uso

de algoritmos genéticos con ellos. El simulador consiste en unos robots simples que exploran un paisaje inventado, recogen muestras y transmiten datos. Los robots reciben órdenes y se coordinan con el resto para realizar tareas más complejas. En este entorno, periodos largos de ejecución simulan misiones de larga duración y es una buena oportunidad para el testing de secuencias largas.

En [62] los autores proponen usar diagramas UML y ACO, en vez de algoritmos genéticos como en [13], para generar datos de tests. Concretamente, un grupo de hormigas explora el grafo inducido por el diagrama de estados UML y trata de generar datos de test óptimos para cumplir el requisito de cobertura de tests considerado. Para poder aplicar ACO a la generación de casos de tests, es necesario:

- Transformar el problema de testing en un grafo (muy habitual en la aplicación de métodos de computación evolutiva para resolver problemas de testing).
- Un test heurístico para medir la *calidad* de los caminos a través del grafo.
- Un mecanismo para crear posibles soluciones eficientemente y un criterio adecuado para finalizar la generación de soluciones.
- Un método apropiado para actualizar las feromonas.
- Y una regla de transición para determinar la probabilidad de que una hormiga vaya de un nodo al siguiente en el grafo.

El juego de tests generado tiene que cumplir tres criterios:

- Cubrir todos los estados.
- Ser viable, es decir, cada caso de test representa un camino posible en el diagrama correspondiente.
- Ser óptimo, el juego de tests no contiene casos de test redundantes y contiene las secuencias de test más cortas posibles.

Se considera el problema de hacer trabajar simultáneamente a un grupo de hormigas para realizar una búsqueda cooperativa en un grafo dirigido G. Una hormiga k en un vértice v del grafo está asociada con una tupla (S_k, D_k, T_k, P) donde S_k almacena los vértices recorridos

por la hormiga; D_k indica aquellos vértices que están siempre conectados al vértice actual v; T_k representa las conexiones directas del vértice actual v con sus vértices vecinos; P representa los niveles de feromona en los vértices vecinos a los cuales puede moverse una hormiga en el vértice actual. S_k, D_k, y T_k son datos privados en cada hormiga, mientras que el conjunto P es un dato compartido por todas las hormigas.

El algoritmo empleado por una hormiga k consiste en: (1) evaluar el vértice actual α, actualizar el camino introduciendo el vértice α en el conjunto S_k y evaluar las conexiones a α para definir T_k usando la tabla de transición de estados asociada al diagrama UML; (2) moverse al siguiente vértice seleccionando el destino dependiendo de la cantidad de feromona depositada, actualizar las feromonas y mover la hormiga.

Existen dos problemas que son habituales en el testing de software basado en estados. Por un lado, algunos de los casos de test generados no son viables y, por otro lado, se tienen que generar inevitablemente muchos casos de test redundantes para conseguir alcanzar todos los estados. No obstante, el algoritmo expuesto posee ciertas ventajas: esta propuesta usa directamente UML estándar creado en el proceso de diseño del software y la secuencia de test generada automáticamente siempre es viable, no redundante y cumple el criterio de cubrir todos los estados.

El propósito en [29] es el empleo de métodos basados en colonias de hormigas para generar un conjunto de tests apropiado para un sistema software. Como novedad respecto a los artículos presentados anteriormente, el diseño y los posibles usos del sistema software son modelados usando un *Modelo de Uso de Markov* (MUM o *Markov Usage Model*) [104, 105] que es una descripción versátil del modelo para el uso de un producto software dado en el campo de la aplicación, que refleja la distribución operacional del sistema. Además, el MUM se enriquece con estimaciones de probabilidades de fallos, pérdidas en caso de fallo y costes de testing. Se parte de la idea de que es muy común que los usuarios no usen todas las funciones descritas en la especificación de un producto con la misma frecuencia. Este hecho da al problema considerado el carácter de un problema de *decisión bajo incertidumbre*. Los autores consideran la dependencia mutua existente entre el cubrimiento y los costes de testing, intentando encontrar un compromiso óptimo entre ambos usando dos variantes del método ACO estándar. El principal problema es que para hallar una solución, el desarrollador no tiene un conocimiento exacto de los posibles fallos y pérdidas en caso de fallo, dichos datos necesitan ser cuantificados para enriquecer el MUM con esta información.

En la literatura ya se han estudiado problemas similares, principalmente en el contexto de testing de protocolos o testing de interfaces de usuario [12, 24, 107]. Si bien dichos trabajos intentan satisfacer algunos criterios de cubrimiento predefinidos (típicamente, cubrimiento de arcos) en esta propuesta no se fuerza un cubrimiento completo. Para aplicar las variantes de ACO, es necesario identificar los estados del software, las posibles transiciones entre los estados y las probabilidades de las posibles transiciones. Es conveniente usar un grafo dirigido con muchas conexiones para la representación del MUM donde los nodos son los estados, las aristas son las transiciones y las probabilidades de las transiciones (estrictamente positivas) se indican con una etiqueta sobre cada uno de los arcos. En el MUM existen un único nodo inicial y un único nodo final, donde además se añade un arco adicional del nodo final al inicial representando una nueva llamada al programa.

En los experimentos presentados, se aplican tres variantes diferentes del algoritmo ACO. En la variante *estándar*, la probabilidad de elegir un arco es proporcional al nivel de rastro ($\tau(e)$) y a un *valor de atracción* ($\eta(e)$). En la *Variante A*, la probabilidad de que una determinada arista e sea elegida es proporcional a $\tau(e) \times \eta(e)$ e inversamente proporcional al número total de veces que la arista e ha sido atravesada en todos los pasos anteriores. De este modo, las aristas que ya han sido usadas frecuentemente toman una probabilidad menor para ser elegidas en el futuro, lo que favorece la diversificación de las rutas. En la tercera variante (*Variante B*) se aplica el mismo mecanismo de selección que en la variante estándar, pero se modifica el mecanismo de actualización del rastro de feromonas: sólo se aumenta el rastro en aquellas aristas que pertenecen a la mejor ruta encontrada hasta el momento cuando el camino atraviesa un mínimo número de aristas de la mejor ruta conseguida. Esta variante también favorece la diversificación.

Como caso de estudio el algoritmo es aplicado a un sistema de tamaño medio del mundo real que representa un centro de llamadas usado por la Cruz Roja austriaca. El MUM considerado consiste en 47 nodos y 85 arcos y refleja 3 menús principales y submenús. La descripción completa ha sido dividida en 4 diagramas: la sección principal y tres subsecciones. El objetivo perseguido es la derivación automática de un conjunto de tests adecuado que recorra el grafo MUM usando un algoritmo ACO.

En el artículo se recogen los datos de las mejores soluciones halladas por los 3 esquemas presentados. La versión ACO estándar encuentra buenas soluciones a partir de 35 iteraciones del algoritmo, mientras que en las 10 - 20 primeras iteraciones los resultados son bastante pobres. Sin embargo, las variantes A y B del método obtienen soluciones relativamente buenas conseguidas en un número pequeño de iteraciones, lo que hace a estos métodos es-

pecialmente interesantes en su aplicación a sistemas más complejos. Todos los algoritmos encuentran soluciones muy razonables después de 70 iteraciones, lo que supone aproximadamente 3 segundos. En estos resultados experimentales se observa que las propuestas mejoran los resultados obtenidos por una solución heurística de tipo devoradora.

En [6] se propone usar un nuevo tipo de modelo ACO para refutar propiedades de seguridad en sistemas concurrentes. Mientras que en los anteriores problemas ACO se aplicaba a la generación de tests [29, 62], en este artículo se aplica al model checking, más específicamente a la refutación de propiedades de seguridad.

En este artículo se trata un problema de model checking en el que se analizan todos los posibles estados del programa para probar (o refutar) que la implementación satisface una propiedad dada. La propiedad se especifica usando una lógica temporal como LTL (Linear Temporal Logic) o CTL (Computation Tree Logic). La herramienta utilizada es SPIN [54] (uno de los model checkers explícitos más conocidos) que toma como entradas un modelo software codificado en Promela y una propiedad especificada en LTL y transforma el modelo y la negación de la fórmula LTL en un *autómata de Büchi* [38] (una extensión de un autómata de estados finitos con entradas infinitas) para realizar el producto síncrono de ambos. Se explora el autómata resultante para buscar un ciclo de estados que contenga un estado de aceptación alcanzable desde el estado inicial. Si se encuentra el ciclo, entonces existe al menos una ejecución del sistema que no satisface la propiedad LTL (para más detalles ver [55]). Si ese tipo de ciclo no existe, entonces el sistema satisface la propiedad y la verificación finaliza con éxito.

En [6] se propone usar un nuevo modelo ACO para grafos muy grandes, para encontrar contraejemplos de fórmulas LTL en sistemas concurrentes. El modelo presentado es capaz de tratar problemas de optimización construyendo un grafo de tamaño desconocido que se construye según avanza la búsqueda. El objetivo consiste en explorar el grafo con una pequeña cantidad de memoria. Para evitar la construcción de soluciones candidatas completas, se limita la longitud del camino recorrido por una hormiga en la fase de construcción. De este modo, la fase de construcción se puede realizar en un tiempo limitado y con una cantidad de memoria limitada. Sin embargo, esta limitación implica que la mayoría de los caminos sean soluciones parciales y que se necesite una función de fitness que pueda evaluar soluciones parciales.

La limitación del camino de las hormigas introduce un nuevo problema: existe un nuevo

parámetro en el algoritmo (λ_{ant}) que establecerá la longitud máxima de los caminos y cuyo valor óptimo no es fácil de establecer a priori. Si se selecciona un valor más pequeño que la *profundidad* (donde la profundidad de un nodo es la longitud del camino más corto desde el estado inicial hasta dicho nodo), el algoritmo no encontrará ninguna solución al problema. Es por ello que se debe seleccionar un valor mayor que la profundidad de un nodo objetivo. Como este valor normalmente se desconoce, se proponen dos alternativas para conocer la profundidad de un determinado nodo: *la técnica de expansión y la del misionero*. En la Figura 3.1 se muestra gráficamente el comportamiento de las dos alternativas propuestas.

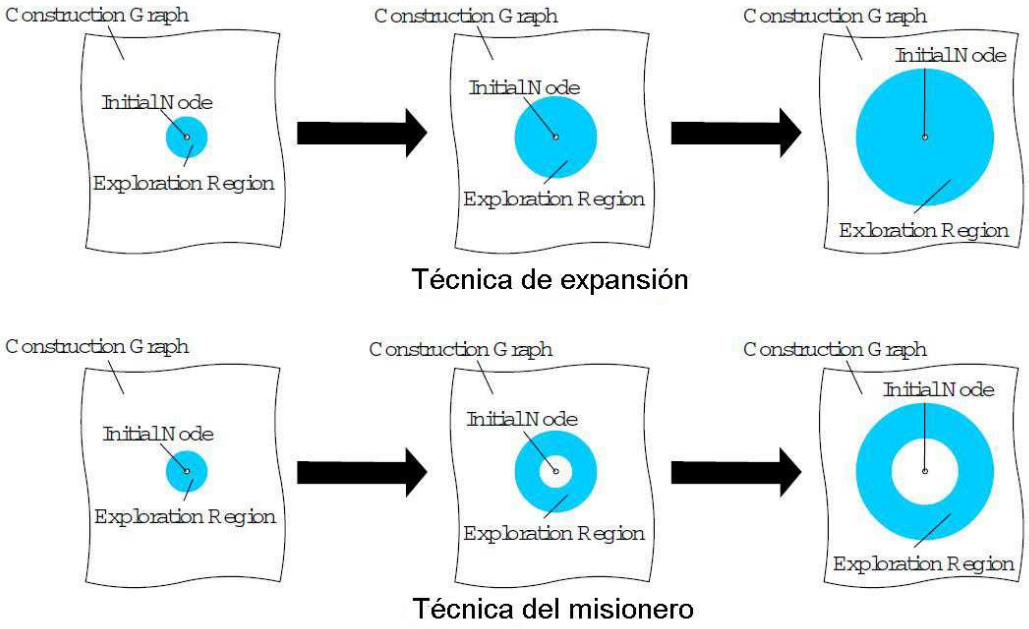

Figura 3.1: Alternativas para alcanzar el nodo objetivo

La primera de las técnicas consiste en ir incrementando dinámicamente λ_{ant} durante la búsqueda mientras que no se encuentre el nodo objetivo. Si la profundidad del nodo objetivo es alta (el caso peor) el comportamiento puede ser el de un método ACO estándar, pero si la profundidad no es muy alta el método puede ser más eficiente. La segunda alternativa consiste en comenzar la construcción del camino de las hormigas desde diferentes nodos durante la búsqueda. Es decir, al principio las hormigas se sitúan en los nodos iniciales del grafo y el algoritmo se ejecuta durante un número dado de pasos σ_s. Si no se encuentra el nodo objetivo, los últimos nodos de los caminos construidos por las hormigas se usan como nodos de inicio para las siguientes hormigas. En el siguiente paso del algoritmo, las nuevas hormigas

comienzan a recorrer el grafo empezando en los últimos nodos de los caminos calculados en el paso anterior. De este modo, la longitud de los caminos se mantiene constante, al igual que la memoria requerida y el tiempo de CPU.

Los resultados muestran que usando una búsqueda heurística se puede reducir tanto la longitud de los contraejemplos como la cantidad de memoria necesaria para obtener un error, permitiendo la exploración de modelos más grandes. Además, la aplicación presentada para resolver el problema de comprobar propiedades de seguridad en sistemas concurrentes muestra que los algoritmos definidos son capaces de mejorar los métodos exhaustivos conocidos en eficacia y eficiencia. Este hecho los hace adecuados para comprobar propiedades de seguridad en sistemas concurrentes grandes, donde las técnicas tradicionales no encuentran errores debido al tamaño del modelo. Por último, estos algoritmos se pueden usar con otras técnicas para reducir la cantidad de memoria necesaria en las búsquedas como la reducción de orden parcial [63], la reducción simétrica [56] o la compresión de estados.

En [40] se presenta un método para explorar espacios de estados muy grandes de sistemas concurrentes reactivos. El esquema presentado utiliza algoritmos genéticos para guiar la búsqueda en el espacio de estados y llegar a estados de error. Para implementar este esquema se emplea Verisoft [39], una herramienta para explorar sistemáticamente el espacio de estados de aplicaciones software compuesta de varios procesos concurrentes que ejecutan código arbitrario.

Las soluciones candidatas a ser consideradas por el algoritmo genético son secuencias finitas de transiciones en el espacio de estados empezando en el estado inicial. Cada solución candidata se codifica usando un cromosoma. Para evaluar la idoneidad de un cromosoma, el algoritmo genético ejecuta el camino codificado por el cromosoma invocando un model checker. Dada la representación de un sistema, el model checker determina el espacio de estados a explorar. La ejecución de un camino comienza en el estado inicial. Si existe más de una posible transición desde el estado actual, el model checker informa al algoritmo genético sobre el número de posibilidades. El algoritmo genético decodifica una parte del cromosoma y la procesa, informando al model checker sobre qué transición escoger. El model checker comprueba si el estado al que lleva dicha transición es un estado de error. Si lo es, se almacena el camino actual y se notifica al usuario. En otro caso, el model checker repite el proceso desde el nuevo estado.

Como un cromosoma sólo puede codificar un número finito de transiciones, el espacio de

estados sólo se explora hasta una profundidad fija, lo que provoca que el algoritmo genético evalúe la idoneidad del cromosoma actual. Una vez que se ha calculado la idoneidad del cromosoma actual, se evalúa otro cromosoma de la población actual usando el mismo proceso.

En este trabajo se pretende detectar dos clases de errores en el espacio de estados: los *deadlocks*, que son estados de los cuales no salen transiciones (todos los procesos del sistema están bloqueados) y las *violaciones de aserciones*, que son los casos en que ciertas expresiones booleanas que involucran a variables del programa se evalúan como falsas. Para guiar la búsqueda genética de ambos tipos de estados de error se definen dos heurísticas diferentes. Para detectar los deadlocks, la heurística para medir la idoneidad de un cromosoma consiste en sumar el número de transiciones posibles en cada estado a lo largo de la ruta de ejecución representada por el cromosoma. Y para detectar violaciones de aserciones, una posible heurística consiste en intentar maximizar la evaluación de aserciones. Para conseguir esto se *premia* a aquellos cromosomas que llevan a las máximas evaluaciones de aserciones posibles.

El problema tratado en este artículo es la exploración de espacios de estados (muy grandes) de sistemas concurrentes reactivos definidos con un model checker, por lo que se requiere el uso de cromosomas novedosos y funciones de fitness adecuadas para la aplicación al campo considerado. En los experimentos realizados se muestra que, para encontrar errores en espacios de estados muy grandes, se puede usar una búsqueda genética utilizando heurísticas sencillas y así conseguir mejorar significativamente los resultados de las búsquedas tradicionales aleatorias y sistemáticas usadas en los actuales model checkers.

En [7] se compara un algoritmo genético con una técnica exacta clásica y se propone el uso de un nuevo operador (llamado *operador de memoria*) para resolver el problema afrontado que permite a los algoritmos genéticos explorar espacios de búsqueda especialmente grandes. Concretamente se emplea una variación en un algoritmo genético combinado con Java Pathfinder [45], un conocido model checker, para buscar errores en aplicaciones complejas. Si bien el objetivo considerado es la búsqueda de *deadlocks*, con el algoritmo propuesto se podría buscar cualquier tipo de violación de propiedades de seguridad cambiando únicamente la función de fitness.

Para detectar caminos que conduzcan a *deadlocks* y que sean preferiblemente cortos, se define la función de fitness como:

$$f = deadlock + numblocked + \frac{1}{1 + pathlen}$$

Donde la variable *numblocked* representa el número de hilos bloqueados generados por el camino; *pathlen* representa el número de transiciones del camino y *deadlock* es 1 si se ha encontrado algún deadlock y 0 en caso contrario. El algoritmo genético tratará de maximizar la función f. Esta ecuación también asume que el número de hilos del modelo a chequear es constante y que un deadlock sólo ocurre cuando todos los hilos están bloqueados. En otro caso, el valor *deadlock* deberá establecerse a un valor mucho mayor cuando de hecho se encuentra un deadlock.

Para encontrar violaciones de propiedades de seguridad, el model checking trabaja buscando por todos los estados de un modelo y validando cada uno de ellos con las propiedades especificadas. Si todos los estados son válidos, queda probado que esas propiedades son válidas en el modelo. Si cualquiera de esas propiedades provoca un error, el model checker puede dar un camino de ejecución que conduce a ese estado de error. En este caso, un individuo (cromosoma) representa un camino a un estado que provoca una violación de una propiedad de seguridad en el modelo. El camino puede ser descrito como una secuencia de transiciones desde el estado inicial hasta el estado final. Como el número de transiciones necesarias para alcanzar el estado objetivo no es conocido de antemano, se usan cromosomas de longitud variable.

El nuevo operador introducido, llamado *operador de memoria*, se utiliza para permitir que un camino se pueda extender indefinidamente mientras se preserva la memoria requerida para evaluar a los individuos. Como la población evoluciona y crece, las primeras transiciones en los individuos tienden a estabilizarse, pero el model checker todavía tiene que evaluarlas en todas las generaciones. Los autores proponen salvar los estados resultantes de esas primeras transiciones en *slots* de memoria y usarlos como puntos de partida para futuras generaciones. Las ventajas son evidentes: es necesaria menor cantidad de memoria y tiempo para evaluar cada individuo y el camino puede mantener un crecimiento constante sin necesitar más memoria. Por supuesto, también existen algunas desventajas: parte del espacio de búsqueda queda descartado y una buena solución podría estar en esa parte.

En los experimentos se trata de encontrar deadlocks en el problema de la *cena de los filósofos* [1] y en el problema de los *matrimonios estables* [3]. En primer lugar se compara el algoritmo genético propuesto usando el operador de memoria y sin usarlo. En el problema de la cena de los filósofos, el algoritmo genético que usa el operador de memoria es más rápido a la hora de encontrar los deadlocks, requiere menos memoria y obtiene mejores resultados. Sin embargo, con el problema de los matrimonios estables se da la situación contraria. El algoritmo genético que no usa el operador de memoria es más rápido en encontrar el camino a

un deadlock y encuentra caminos más cortos y con mejores resultados. Al usar el operador de memoria se empleaba menor cantidad de memoria, pero se encontraban más dificultades para encontrar un deadlock. El problema es que, al usar el operador de memoria, se seleccionan las mejores soluciones de cada tramo y luego se continúa buscando la mejor solución partiendo de los primeros segmentos de la solución, pero esto impide seleccionar una mala solución parcial al comienzo para encontrar una buena solución global. Es posible que utilizando una mejor función de fitness, en vez de usar la genérica, se solucionasen estos problemas.

También se compara el algoritmo desarrollado con algoritmos estándar de búsqueda exhaustiva (búsqueda primero en profundidad y búsqueda primero en anchura). Se muestra que, generalmente, el algoritmo genético necesita más tiempo que los algoritmos estándar para alcanzar un error, pero las trazas de error generadas por dicho método son más pequeñas y simples, lo que facilita su uso en la depuración del programa. Además, el método puede usarse en problemas más grandes, donde los métodos estándar no se pueden emplear debido a la cantidad de memoria necesitada. Sin embargo, existen varios problemas en el método presentado. Un problema es el hecho de que se debe limitar el crecimiento de los cromosomas de los individuos. Esto requiere tener a priori una estimación de la longitud de la cadena de error, que en principio es imposible conocer. El otro problema es la cantidad necesaria de memoria en este algoritmo, al menos en su versión estándar. Por ello se propone el operador de memoria, mediante el cual se podrían resolver estos problemas e incluso manejar problemas más grandes.

Finalmente se propone modificar el algoritmo propuesto para detectar propiedades de *viveza*. Esto implicaría añadir una fase más al algoritmo en la que, en vez de buscar un estado de error, se busque un ciclo que incluya el estado encontrado en la primera fase.

En [49] los autores presentan los requerimientos necesarios para crear un método formal integrado para el análisis de sistemas basados en nubes de partículas. Futuras misiones de la NASA explotarán nuevos paradigmas para la exploración del espacio enfocado a las emergentes tecnologías de los sistemas autónomos y automáticos. Los conceptos de las misiones tradicionales están siendo complementados con nuevos conceptos que involucran a muchas naves espaciales pequeñas, trabajando de forma cooperativa, análogamente a los enjambres en la naturaleza. Esto ofrece varias ventajas: la capacidad de mandar naves a explorar regiones del espacio donde las naves tradicionales no podrían llegar y la reducción de los costes y el peligro de las misiones. El grado de autonomía de estas misiones haría necesaria una canti-

dad prohibitiva de testing para asegurar la corrección del sistema. Además, el aprendizaje y la adaptación implican una continua mejora en el comportamiento y la aparición de nuevos patrones de comportamiento que no pueden ser predichos usando métodos tradicionales de desarrollo. El resultado es que las técnicas de especificación y de verificación formal jugarán roles vitales en el desarrollo futuro de las misiones de exploración del espacio de la NASA.

Así como el software empleado en las misiones espaciales se vuelve más complejo, también se vuelven más difíciles las tareas de testing y de búsqueda de errores. Esto es especialmente cierto en procesos altamente paralelos y sistemas distribuidos, y ambas son características de los sistemas basados en nubes de partículas. Las condiciones de ejecución en estos sistemas normalmente no se pueden encontrar introduciendo datos de ejemplo y chequeando si los resultados son correctos. Estos tipos de errores dependen del tiempo y sólo ocurren cuando los procesos envían o reciben datos en un tiempo en particular o en una secuencia en particular. Para encontrar estos errores usando testing, el proceso de software tiene que ejecutarse en todas las posibles combinaciones de estados en los que los procesos podrían estar a la vez en él. Como el espacio de estados es exponencial con el número de estados, no se puede testear con un número relativamente pequeño de elementos en la nube de partículas. Tradicionalmente, para tratar el problema de la explosión de estados, los verificadores reducen artificialmente el número de estados del sistema y aproximan el comportamiento del software usando modelos.

Se ha probado que los métodos formales mejoran el correcto funcionamiento de complejos sistemas formados por componentes que interactúan entre sí. Una vez escrita, se puede usar una especificación formal para probar propiedades de un sistema, chequear distintos tipos de errores y usarse como entrada de un model checker. La verificación de comportamientos emergentes es un área que, desafortunadamente, no tratan adecuadamente la mayoría de métodos formales. Por ello surge el *proyecto FAST*, que es un proyecto que estudia técnicas de métodos formales para determinar si algún método formal es adecuado para verificar sistemas basados en nubes de partículas y su comportamiento evolutivo. También trata de identificar una propuesta formal integrada para tecnologías basadas en nubes de partículas. El proyecto encontró sólo dos propuestas formales que, con algunas modificaciones, pudieran analizar el comportamiento evolutivo de los *enjambres*. WSCCS (Weighted Synchronous Calculus of Comunicating Systems) [99, 98], una álgebra de procesos usada para analizar aspectos sociales de los insectos; y las *Máquinas-X* (en inglés X-Machines) [52, 53] que han sido usadas para modelar biología celular y que con algunas modificaciones pueden especificar nubes de partículas. Sin embargo, estas propuestas no predicen comportamientos evolutivos del modelo, pero sí el comportamiento después de la evolución.

El proyecto ha definido un método formal integrado que es apropiado para el desarrollo de los sistemas estudiados. Las propiedades necesarias para la especificación y la predicción del comportamiento de las nubes de partículas son las siguientes: representación de procesos, razonamiento, elección de acciones alternativas, envío de mensajes asíncrono, almacenamiento o buffering de mensajes, determinación de si se han logrado los objetivos, métodos para determinar nuevos objetivos y predecir comportamientos evolutivos entre otros. En el proyecto seleccionaron cuatro métodos formales que combinados servirán para especificar nubes de partículas: las álgebras de procesos CSP [50, 48] y WSCCS [99, 98], las Máquinas-X [51] y la Unidad Lógica [71]. Se concluye que la integración de estos métodos es la mejor técnica para la verificación de sistemas cooperativos basados en nubes de partículas. La integración de los aspectos de memoria y funciones de transición de las Máquinas-X con prioridades y la información probabilística de WSCCS y los otros métodos produce un método de especificación que previsiblemente permitirá la construcción de las herramientas necesarias para especificar el comportamiento evolutivo de los sistemas basados en nubes de partículas.

En [100] se muestran los resultados de una investigación referente al desarrollo de un programa de generación automática de datos de tests para respaldar el testing de sistemas software de seguridad crítica. Para ello se explotan técnicas de búsqueda independientes que permitan el uso de nuevos criterios de test. Estos criterios son dirigidos mediante funciones que cuantifican lo adecuados que son los datos de test según algún criterio previamente establecido. El centro del proyecto consiste en investigar técnicas para validar un software de un controlador digital de aviones (FADEC, Full Authority Digital Engine Controller). Este software está sometido a gran cantidad de cambios y los costes asociados a la re-verificación son muy elevados. El objetivo consiste en desarrollar un soporte automático de testing para reducir dichos costes manteniendo o mejorando la calidad. Las técnicas desarrolladas son también aplicables a otros sistemas software de seguridad crítica. La estructura está basada en la aplicación de diferentes técnicas de búsqueda, a diferencia de los anteriores trabajos presentados. En particular, se puede definir un criterio de test simplemente proporcionando una función de fitness que dé una medida cuantitativa de la idoneidad de cada criterio de test.

El software de seguridad crítico se necesita testear, ya que un fallo en estos sistemas puede provocar heridos o muertos. Además se introducen algunas propiedades de interés durante el testing, ya que las actividades de verificación tradicionales se centran en la corrección

funcional del software, lo que no es suficiente para sistemas de seguridad crítica. Otro de los puntos novedosos del trabajo es que se utilizan dos tipos de análisis para verificar propiedades de interés en el sistema: estático y dinámico. El *análisis estático* involucra la construcción y análisis de un modelo abstracto matemático del sistema, pero no ejecuta el software que se quiere testear. El *análisis dinámico*, sin embargo, involucra la ejecución del software y monitoriza su comportamiento.

El sistema propuesto implementa las siguientes heurísticas: búsqueda aleatoria, escalada por máxima pendiente, enfriamiento simulado y algoritmos genéticos. Todas estas técnicas se utilizan para la generación de distintos juegos de tests. El sistema desarrollado en el proyecto se ha desarrollado de tal modo que sea extensible para la generación de nuevos datos de test. Para la selección de los datos de test, y consecuentemente su generación, hay que buscar datos de test según un criterio particular. Normalmente el verificador del software es el responsable de encontrar los mejores datos de test dado algún criterio con algunas restricciones (a menudo de tiempo o de dinero). Sin embargo, este proceso puede llevar mucho tiempo, ser difícil y ser caro. A menudo, el verificador debe crear un conjunto de tests para ejecutar una parte determinada del sistema. La función de fitness deberá guiar a los métodos de búsqueda para automatizar esta generación de tests. En algún punto de la ejecución del sistema que pertenezca a la parte del sistema a analizar, se invoca a la rutina de la función de fitness para evaluar la idoneidad de los datos de test actuales. Esta función evaluará el valor de todas las variables locales en un punto específico de la ejecución. Concretamente, en el algoritmo genético para la generación de datos de test, se implementa el operador de mutación de distintos modos:

- Mutación simple: establece un parámetro con otro valor válido generado de manera aleatoria.
- Mutación aleatoria: reemplaza una solución entera por una solución generada aleatoriamente.
- Mutación de vecindario: utiliza el enfriamiento simulado. Los valores de los parámetros se sustituyen por valores *vecinos* de los actuales.

El sistema desarrollado soporta la generación de datos de test explicada. Consiste en una herramienta de extracción de información - Figura 3.2 -, una colección de generadores de funciones de fitness y la implementaciones de las técnicas de búsqueda. *La herramienta de extracción de información* toma el sistema a testear y el criterio de test deseado y extrae la información que necesitan los generadores de funciones de fitness. *Los generadores*

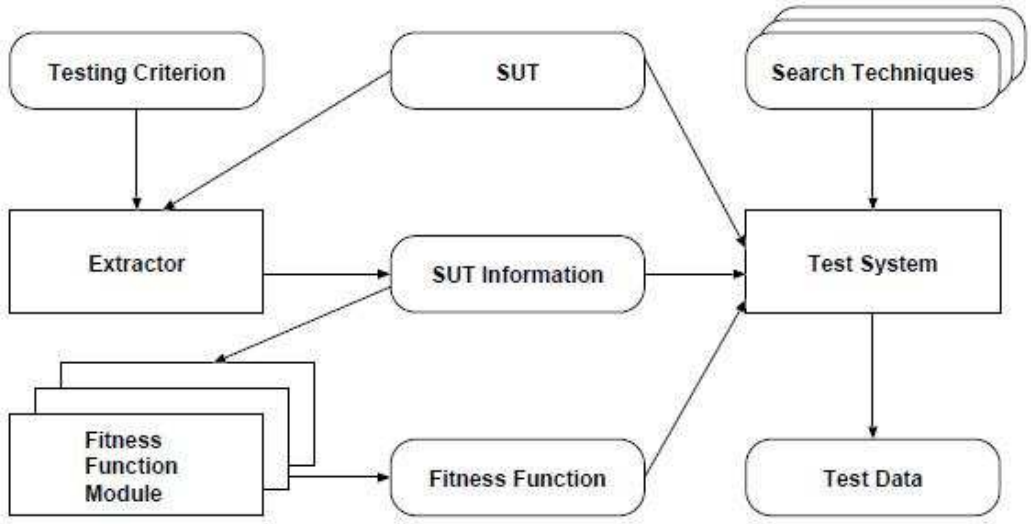

Figura 3.2: La herramienta basada en búsquedas

de funciones de fitness toman la información extraída del sistema y generan un código para evaluar la efectividad de los datos de test dependiendo del criterio de test seleccionado. Por último, se compilan juntos: la función de fitness, la implementación de la *técnica de búsqueda empleada* y el sistema, para así formar el sistema de generación de casos de test. Al ejecutar el programa resultante de dicha compilación, se inicia el proceso de generación de datos de test usando la técnica de búsqueda seleccionada para localizar datos de test que cumplan el criterio de test especificado. Se emplean las técnicas de búsqueda para localizar valores óptimos de la función de fitness.

A continuación exponemos brevemente otros trabajos relacionados. Los algoritmos genéticos ya han sido usados en un amplio rango de aplicaciones. En particular, los algoritmos genéticos se han usado para realizar testing estructural y funcional de programas secuenciales. Por ejemplo, en [75] se presenta una técnica para la generación automática de datos de test usando un algoritmo genético guiado por un programa de control de dependencias; su implementación está dirigida a conseguir cubrimiento de ramas y de declaraciones (statement and branch coverage). En [57] se usan algoritmos genéticos para generar conjuntos de tests que satisfagan los requisitos de un conjunto de datos de test adecuado para testing estructural. En [18] se utiliza como punto de partida el trabajo [57] y se presenta una herramienta para

la automatización de la generación de datos de test y la identificación de caminos inviables. Mientras que en [59] se desarrolla un marco de trabajo que usa algoritmos genéticos para que los métodos de testing puedan manejar estructuras de datos complicadas; este método fue aplicado satisfactoriamente para identificar diferentes errores en una arquitectura de redes dinámicas de dispositivos y ordenadores. Otros trabajos modernos aplican algoritmos genéticos para encontrar juegos de tests adecuados. En particular, en [26] se usan técnicas de *testing de mutación* para evaluar la aptitud de juegos de tests en programas Haskell o en [79] se utiliza un método basado en la formación de los ríos para encontrar tests cortos que alcancen algunos estados o aristas en una FSM asumiendo que se puede guardar y restaurar un estado del sistema.

Capítulo 4

Resumen de los Artículos Originales de la Tesis

En este capítulo presentamos de forma resumida las publicaciones originales que conforman la tesis. Después de introducir dichos artículos, se incluyen también los resúmenes de dos artículos no publicados, actualmente en proceso de revisión en revistas, que mejoran y extienden las aportaciones de dos de las publicaciones incluidas en la tesis.

4.1. Using River Formation Dynamics to Design Heuristic Algorithms

En [80] introducimos por primera vez RFD, el algoritmo heurístico central de esta tesis. Recordemos que el esquema general de este algoritmo se describe en detalle en la sección 2.6. Es por ello que la descripción de esta sección se centrará en presentar la motivación general del algoritmo y en describir la experimentación con el mismo desarrollada en [80].

Como ya vimos en el capítulo 2, los métodos inspirados en la naturaleza son útiles porque ofrecen un punto de vista alternativo para afrontar problemas que son especialmente difíciles. Este es el caso de los problemas NP-duros, que se supone que, en el peor de los casos, necesitan tiempo exponencial para ser resueltos. Como en la práctica no es posible resolver estos problemas, se sustituyen los métodos óptimos por algoritmos heurísticos que, normalmente, necesitan tiempo polinómico para obtener soluciones subóptimas. En este artículo consideramos el *problema del viajante de comercio* (en adelante TSP, del inglés Traveling Salesman Problem). Este problema NP-completo ha atraído la atención de muchos desarrolladores debido a que, por un lado, encontrar caminos óptimos es un requisito que

aparece a menudo en aplicaciones reales y, por otro lado, es un marco adecuado para probar métodos heurísticos.

Un método que proporciona eficientes algoritmos heurísticos para resolver el TSP es la optimización basada en colonias de hormigas (ACO). Como expusimos en la sección 2.5, este método consiste en copiar el método usado por las hormigas para encontrar buenos caminos desde su nido hasta las fuentes de comida. Sin embargo, estos métodos también tienen algunos problemas intrínsecos. Algunas veces, los caminos formados por las hormigas pueden interferir entre sí y dañarse mutuamente. Por ejemplo, se pueden crear y reforzar ciclos locales. Otro de los problemas asociados a estos métodos es que si una hormiga encuentra un camino mejor, se necesita que muchas hormigas tomen ese camino y refuercen el rastro de feromonas para que esa ruta se establezca como la favorita frente a otros caminos antiguos ya establecidos.

Ambos problemas son consecuencia de la siguiente propiedad: cuando una hormiga tiene que decidir su próximo movimiento, sólo tiene en cuenta el rastro de feromona en cada uno de los posibles destinos, ignorando el rastro presente en el origen. Ahora supongamos que, en lugar de basar los movimientos de las hormigas en la cantidad de feromonas en cada posible destino, consideramos la *diferencia* de feromonas entre el origen y el destino, requiriendo que esta diferencia sea positiva, es decir, el rastro de feromona debe ser mayor en el destino que en el origen. Esta alternativa ayuda a superar los problemas expuestos. En particular, como vimos en la sección 2.6, el uso de diferencias de feromonas hace que los ciclos locales resulten contradictorios y provoca la rápida preferencia por los caminos más cortos. Pero ahora surge una pregunta: ¿cómo hacer que los rastros de feromonas sean crecientes en cada uno de los caminos? Para ello, dejamos de lado la metáfora de las hormigas y consideramos otro marco de trabajo. El marco de trabajo alternativo también está basado en la naturaleza, en particular en la *formación dinámica de los ríos*, y simula el proceso geológico de la lluvia, la erosión y la sedimentación. Presentamos un algoritmo basado en estas ideas llamado RFD (del inglés River Formation Dynamics), estudiamos cómo este método permite resolver los problemas que aparecen en los métodos ACO y, además, aplicamos el algoritmo para resolver el TSP, aunque para ello es necesario adaptar el esquema general.

Los resultados obtenidos son comparados con los resultados obtenidos por un método ACO y concluimos que el método ACO obtiene mejores soluciones en los primeros segundos de ejecución ya que las hormigas consiguen converger más rápidamente. Sin embargo, cuando pasa algún tiempo más, la calidad de las soluciones del método RFD mejora la calidad de las soluciones halladas por ACO debido a que la exploración realizada por nuestro método

es más profunda.

4.2. Finding Minimum Spanning/Distances Trees by using River Formation Dynamics

En [81] adaptamos el esquema general del método RFD presentado en [80] para tratar con dos nuevos problemas. Estos problemas consisten en, dado un grafo valorado, (1) encontrar el árbol recubridor mínimo y (2) encontrar el árbol de distancias mínimo, es decir, un árbol tal que la suma de las distancias desde cada nodo a un *nodo de salida* dado es mínima. La forma estándar de estos problemas no requiere el uso de métodos heurísticos porque se pueden resolver polinómicamente, por ejemplo usando los algoritmos de Kruskal y Dijkstra respectivamente. Sin embargo, algunas generalizaciones de estos problemas son NP-completos. Introducimos la siguiente generalización: consideramos que el coste de tomar una arista e depende del camino seguido hasta entonces. Es decir, si se atraviesa e después de haber seguido un camino σ, entonces el coste de añadir e al camino es $c_{e,\sigma}$; en general, $c_{e,\sigma} \neq c_{e,\sigma'}$ para cualquier otro camino σ'.

En el artículo se añade una definición formal de los nuevos problemas introducidos y también se define un *grafo de coste variable*. El problema del árbol de distancias mínimas en un grafo de coste variable, que denotamos por MDV, consiste en lo siguiente: dado un grafo de coste variable G y un número natural $K \in \mathbb{N}$, encontrar un árbol G' en G tal que el *coste de distancias* (suma de costes desde ciertos nodos hasta un nodo determinado, ver [81]) de G' sea menor o igual que K. Y el problema del árbol recubridor mínimo en un grafo de coste variable, que denotamos por MSV, consiste en lo siguiente: dado un grafo de coste variable G y un número natural $K \in \mathbb{N}$, encontrar un árbol G' en G tal que el *coste de recubrimiento* (suma de costes promediados del árbol, ver [81]) de G' sea menor o igual que K.

La generalización de los problemas presentados aumenta su aplicabilidad a nuevos escenarios. Por ejemplo se puede aplicar a problemas de testing, donde uno de los objetivos perseguidos es interactuar con un sistema de tal modo que todos los estados se alcancen al menos una vez, problema que se ajusta al MSV.[1] Por otro lado, el problema MDV permite abordar la construcción de determinados tipos de redes de área local (LAN). En particular,

[1] Dicho problema de testing se abordará explícitamente en el artículo [87] (ver la sección 4.6 dentro de un marco de *máquinas de estados finitos*. El problema MSV, al incluir explícitamente el uso de variables, extiende dicho problema de testing al contexto de las *máquinas de estados finitos extendidas*, que incluyen el uso de variables, aunque no considera explícitamente la posibilidad de recuperar estados anteriores, como de hecho sí hace en [87].

MDV permite que los costes de transmisión dependan de la información que se transmite.

De nuevo aplicamos los algoritmos RFD y ACO para resolver los problemas presentados y comparar las soluciones calculadas. Observamos que las soluciones obtenidas usando ACO son mejores al principio de la ejecución. Sin embargo, tras ejecutar los algoritmos algo más de tiempo, las soluciones halladas por RFD mejoran las obtenidas por ACO. Esto nos muestra la tendencia general de nuestro método a realizar exploraciones más profundas del grafo analizado.

4.3. Solving Dynamic TSP by using River Formation Dynamics

En [86] resolvemos un problema muy relacionado con los problemas de *routing*: el problema del viajante de comercio (TSP) dinámico, en el que los nodos (o hosts) y las aristas (o conexiones) pueden aparecer y/o desaparecer a lo largo del tiempo. Esto permite representar escenarios donde las redes pueden cambiar y obliga a los algoritmos de routing a adaptarse a esos cambios. Resolver el TSP dinámico para encontrar caminos óptimos en una red requiere enfrentarse a una doble dificultad: (1) por un lado se está resolviendo un problema NP-duro y, (2) por otro lado, los datos necesarios para tomar las decisiones están distribuidos, siendo necesario un algoritmo que tome decisiones de acuerdo a información local.

Los métodos de computación evolutiva proporcionan algoritmos que afrontan ambos problemas de manera intrínseca. Es por ello que en este artículo adaptamos el método RFD para resolver el TSP en redes dinámicas. Este problema tiene aplicaciones, por ejemplo, en el routing de redes defectuosas o congestionadas y en la planificación del tráfico. Para afrontar este nuevo problema, identificamos y reforzamos las características de RFD que mejoran su adaptación a los cambios en grafos. La geología nos da algunas características que son importantes en este sentido. Destaquemos que el proceso de erosión proporciona un método para *penalizar* caminos ineficientes así cómo para evitar caminos bloqueados, ya que si un camino conduce a un nodo que está a menor altura que el resto de nodos adyacentes, entonces la gota depositará sus sedimentos incrementando la altura de dicho nodo. Antes o después, la altura de este nodo alcanzará la de sus nodos vecinos permitiendo que gotas posteriores puedan seguir avanzando en su camino. De este modo, los caminos no se verán interrumpidos, obteniéndose un método implícito para evitar comportamientos incorrectos de las gotas, así como para encontrar rutas alternativas cuando un camino antiguo se interrumpe por la desaparición de un nodo o de una arista.

En los experimentos realizados, primero calculamos una solución de una instancia del problema aplicando tanto ACO como RFD. En el siguiente paso introducimos alguno de los siguientes cambios en el grafo: (1) borramos una arista que forma parte tanto de la solución encontrada por RFD como por ACO; (2) borramos un nodo del grafo; (3) añadimos un nuevo nodo. Una vez que se ha introducido el cambio, recalculamos las soluciones con ambos métodos. Cabe destacar que no se reinician los algoritmos, sino que se mantiene el estado actual de los métodos, es decir, la cantidad de feromonas presente en cada arista y la altura de los nodos. En los resultados se observa que el RFD tiene mayor capacidad de reacción de reconstruir buenas soluciones después de un cambio. Cuando se introducen muchos cambios a la vez, los resultados obtenidos son similares: al principio de la ejecución las hormigas consiguen mejores resultados pero las gotas consiguen mejorar sus soluciones tras pasar un poco de tiempo.

4.4. Applying River Formation Dynamics to Solve NP-Complete Problems

En [83] recopilamos todo el trabajo realizado hasta ese momento (2008) y lo ampliamos. Recordemos que el algoritmo RFD fue inicialmente presentado en [80], donde aplicamos este método evolutivo para resolver un problema clásico NP-completo como es el problema del viajante de comercio (TSP) donde era necesario adaptar el esquema general. Después, en [81] estudiamos la aplicabilidad de RFD a otros problemas NP-completos. En particular, aplicamos RFD para resolver los problemas de encontrar el árbol de recubrimiento mínimo y el árbol de distancias mínimas en un grafo con costes variables (MSV y MDV respectivamente), lo que nos permitió analizar la capacidad de RFD para resolver dichos problemas. En ambos artículos, analizamos el rendimiento de RFD para resolver el TSP y el MSV comparando los resultados con las soluciones obtenidas empleando una implementación de ACO. Los experimentos realizados mostraron que el tiempo necesario por RFD para encontrar buenas soluciones es, en general, mayor que el tiempo necesitado por ACO para encontrar soluciones equivalentes, aunque las soluciones encontradas por RFD mejoran las soluciones de ACO tras dejar pasar algo más de tiempo. Esto es debido a que RFD realiza una exploración más profunda del grafo.

En este capítulo del libro *Nature-Inspired Algorithms for Optimisation* resumimos nuestro trabajo previo, mostrando las principales ideas de nuestro método y experimentos anteriores. Además, realizamos nuevos experimentos donde comparamos RFD y ACO para las mismas

instancias del problema MDV. También añadimos nuevos experimentos en los que estudiamos la capacidad de RFD y ACO para tratar con grafos *dinámicos*. En [86] ya se estudió la aplicación de RFD y ACO al problema del TSP dinámico. Ahora estudiamos la aplicación de MSV y MDV sobre grafos dinámicos, lo que nos permitió comprobar la aptitud de las gotas y las hormigas para adaptarse dinámicamente y encontrar caminos en un grafo en el que los nodos y las aristas pueden aparecer y/o desaparecer.

Los experimentos en los casos dinámicos muestran que en algunos casos ACO no consigue adaptarse a los cambios y no encuentra soluciones al problema. Para que ACO encontrase una solución, podríamos reiniciar ACO tras cada cambio con los datos del nuevo grafo, pero este problema deja patente que la adaptabilidad de ACO a ambientes dinámicos es peor que la adaptabilidad de RFD. Teniendo en cuenta aquellos casos en los que ambos métodos hallan una solución, observamos que, de nuevo, ACO encuentra mejores soluciones más rápidamente. Sin embargo, una vez más, la calidad de las soluciones encontradas por RFD es mejor cuando los algoritmos se ejecutan durante algo más de tiempo. Otra nueva contribución en el capítulo del libro es la inclusión de las demostraciones de que los problemas MSV y MDV son NP-completos. Las demostraciones consisten en la reducción polinómica de un problema NP-completo conocido, 3-SAT, a cada uno de los problemas considerados.

4.5. Hybridizing River Formation Dynamics and Ant Colony Optimization

En [85] desarrollamos un método híbrido ACO-RFD tratando de obtener las mejores características de cada uno de los métodos. Para poder manejar ambos métodos, los nodos de los grafos almacenarán un valor de altura, así como cada arista tendrá un valor del rastro de feromona presente. En este nuevo grafo liberamos las entidades híbridas *hormiga-gota*. Estas nuevas entidades contienen todos los atributos necesarios tanto para definir una hormiga como para definir una gota. Cuando una hormiga-gota tiene que decidir su próximo movimiento dentro del grafo, tanto los rastros de feromonas como los valores de altura tendrán un *peso* en dicha decisión. Nótese que los pesos de cada método no son necesariamente fijos a lo largo de toda la ejecución, sino que pueden variar. Después de cada movimiento de la nueva entidad se actualizarán los valores de los rastros de feromonas y las alturas de los nodos de acuerdo a cada uno de los métodos (como se haría en los métodos estándar). Así, este método híbrido se verá influenciado por algunos valores tomados *tal y como son* (los rastros de feromonas) y también por algunos valores *derivativos* (las *diferencias* de alturas).

Para comprobar el funcionamiento del nuevo método híbrido consideramos un problema NP-duro, en particular MDV (el problema de encontrar el árbol de distancias mínimo en un grafo con costes variables). El objetivo de hibridar RFD y ACO en este artículo consistirá en mejorar los resultados obtenidos por ambos métodos. Tras realizar experimentos podemos concluir que el método híbrido obtiene mejores soluciones que los métodos RFD y ACO estándar por separado. La mejora de la calidad de las soluciones es debida a la forma en la que se analiza el espacio de estados. Combinando los dos métodos evitamos que el método híbrido se quede *estancado* en soluciones que son óptimos locales para uno de los métodos pero no para el otro. La alternancia en el peso de cada método durante la ejecución permite que cada método sea el dominante durante cierto tiempo. De este modo, comprobamos que los métodos son compatibles y que pueden colaborar juntos en la formación de buenas soluciones.

4.6. Testing Restorable Systems by using RFD

En [79] buscamos aplicar explícitamente nuestro algoritmo a los métodos formales, en particular a un problema de testing. Como vimos en el capítulo 3, las técnicas formales de testing permiten realizar tareas de testing de un modo sistemático y semi-automático. Normalmente, los testeadores de sistemas no persiguen la *completitud*, sino que aplican algún tipo de *criterio de cobertura*. Por ejemplo, dada una especificación representada por una máquina de estados finitos (FSM, del inglés Finite State Machine), podríamos estar interesados en aplicar un juego de tests que permita alcanzar todas las transiciones o todos los nodos definidos en la especificación.

En este artículo, generalizamos los métodos de testing que tratan de alcanzar algún/todos los estados o transiciones al caso en que el testeador del sistema puede *restablecer* cualquier configuración previa del sistema. Asumimos que la implementación bajo test (IUT, del inglés implementation under test) es un sistema software y que el testeador puede *guardar* la configuración completa actual del sistema en cualquier momento. Más tarde, el testeador podría restablecer dicha configuración y ejecutar el sistema desde ese punto en adelante. En particular, tras restaurar una configuración, podría seguir un camino diferente al que se siguió la primera vez que se ejecutó. Destaquemos que si podemos guardar/restaurar configuraciones, entonces podríamos utilizar esta característica para *evitar repetir* algunas secuencias de ejecución. De esta forma, podríamos ahorrar parte del tiempo asignado a las actividades de testing. Sin embargo, las operaciones de guardar/restaurar una configuración completa de un sistema podrían ser costosas en tiempo. Es por ello que, en general, las

operaciones de guardar/restarurar sólo deberían usarse en aquellos casos en los que el coste de repetir un camino es mayor que el coste de guardar y restaurar una configuración. Este es el caso de muchos sistemas software que realizan operaciones costosas pero que no consumen mucha memoria (por ejemplo, sistemas embebidos o sistemas de control).

En el artículo presentamos un método que, dado (1) el coste de guardar/restaurar una configuración; (2) una especificación FSM que indica explícitamente el coste de realizar cada transición; y (3) un conjunto de configuraciones *críticas* del sistema que se desean testear, devuelve un plan para interactuar con el sistema que permite alcanzar todas las configuraciones críticas en un tiempo reducido. Denotaremos el problema de encontrar la secuencia de interacción *óptima* cumpliendo estas condiciones como el problema de la *Secuencia de Carga Mínima* (MLS, del inglés Minimum Load Sequence). Identificamos que este problema es NP-completo, por lo que es razonable resolverlo de manera heurística. En particular, utilizamos el método RFD para aproximar el camino óptimo. Cabe destacar que en un plan de interacción en el que se permite guardar/restaurar configuraciones, cada punto de restauración representa una *bifurcación*. Es por ello que los planes de interacción encontrados por nuestro método se representarán por un *árbol* que recorre todos los puntos críticos del sistema, donde la raíz es el estado inicial de la FSM. Por ello, nuestro objetivo será encontrar un árbol donde la suma de los costes de todas las transiciones del árbol (incluidas las sumas de los costes de las operaciones de guardar/restaurar) sea mínima.

En los experimentos preliminares realizados (en los cuales sólo se tratan configuraciones críticas que sean nodos, sin tratar configuraciones críticas que sean aristas) se puede observar cómo podemos obtener planes de testing que necesitan menos tiempo cuando se utilizan las operaciones de guardar/restaurar. Además, comparamos los resultados obtenidos por RFD con los resultados obtenidos por un método de ramificación y poda y observamos que las soluciones encontradas por RFD superan de forma clara las soluciones halladas por dicho método de ramificación y poda.

4.7. A Formal Approach to Heuristically Test Restorable Systems

En [87] tomamos como punto de partida el trabajo realizado en [79] e introducimos una definición formal del problema. En particular, introducimos la máquina de estados finitos extendida usada para definir la especificación del sistema (llamada *Máquina de Estados Finitos Valorada* o WFSM, del inglés Weighted Finite State Machine), donde denotamos el

coste de restaurar un estado previo y el coste de las transiciones; definimos el concepto *secuencia de cargas* y el coste asociado a dicha secuencia, teniendo en cuenta los costes de guardar/restaurar una configuración; enunciamos formalmente el problema MLS; también definimos el concepto *árbol de cargas* y el coste asociado a dicho árbol; por último, demostramos que un árbol de cargas t tiene asociada una secuencia de cargas s equivalente cuyos costes son iguales, justificando que es indiferente encontrar una secuencia de cargas mínima o un árbol de cargas mínimo en el problema MLS.

Para aplicar RFD en la resolución del problema MLS, debemos adaptar el esquema general. La raíz del árbol que queremos formar será el estado inicial de la WFSM. En RFD dicho nodo será el nodo destino. Para que las gotas puedan llegar al nodo destino, cada transición de la WFSM será representada por una arista en el sentido contrario en el grafo empleado por el algoritmo. Es necesario el uso de *nodos barrera*, al igual que cuando resolvíamos el problema TSP, así como el uso de aristas adicionales que unan cada nodo con el resto a través del *camino más corto* para permitir la repetición de estados en la solución. Estas aristas adicionales sirven para denotar que se ha vuelto a un estado ya atravesado tomando transiciones repetidas, en vez de cargando. La segunda modificación principal es referente a los costes de carga. Se introduce un incentivo negativo a la formación de puntos de bifurcación proporcional al coste de carga, porque estos puntos representarán *restauraciones* en nuestra solución e implican un coste adicional.

En la sección experimental hemos ampliado las pruebas realizadas en [79] confirmando las conclusiones extraídas. Las soluciones encontradas por RFD son siempre mejores que las soluciones encontradas por un método de ramificación y poda; y permitir cargar un estado atravesado previamente reduce el tiempo necesario para cubrir todos los puntos críticos en grafos poco densos (de hecho, los grafos empleados en las especificaciones definidas usando FSMs son típicamente poco densos). Sin embargo, en grafos densos no es útil, pues es relativamente fácil encontrar caminos que recorran todos los puntos críticos sin necesitar ninguna carga.

4.8. Applying RFD to Construct Optimal Quality-Investment Trees

Aunque RFD ya ha sido aplicado a problemas NP-duros de diferente naturaleza, como el *problema del viajante de comercio* [80] y el *problema de la secuencia de carga mínima* [87], hemos observado que se comporta particularmente bien en problemas que consisten en crear

algún tipo de *árbol de cobertura* sobre un grafo dado. En estos problemas el objetivo es construir un árbol que recorra algunos nodos de tal modo que se satisfaga una propiedad o se maximice/minimice un valor. Adaptar RFD a estos problemas es natural al método: se asigna altura cero a uno de los nodos que debe ser cubierto (representará el mar) y depositamos gotas en el resto de nodos a cubrir. Tras ejecutar cierto tiempo el algoritmo, la gravedad hace que las gotas formen un *árbol* de afluentes desde los puntos de partida hasta el nodo que representa el mar.

En [82] tratamos de explotar las mejores características de RFD para resolver el siguiente problema: dado (1) un grafo valorado, donde se asignan costes a las aristas; (2) un subconjunto de nodos que hay que cubrir llamados *nodos origen*; y (3) un nodo específico llamado *nodo destino*, construimos un árbol tal que (i) los caminos que conectan cada uno de los nodos origen con el nodo destino a través del árbol son tan cortos como sea posible y (ii) el coste del árbol construido es tan pequeño como sea posible. Cabe destacar que los árboles que son óptimos respecto a (i) no tienen porqué ser óptimos respecto a (ii) y viceversa. De este modo, el objetivo consistirá en encontrar un equilibrio entre ambos objetivos. Si X es la suma de los costes desde cada nodo origen hasta el nodo destino a través del árbol e Y es el coste del árbol (es decir, la suma de los costes de las aristas que forman el árbol), entonces nuestro objetivo será minimizar la expresión $\alpha \cdot X + (1 - \alpha) \cdot Y$, donde $0 \leq \alpha \leq 1$.

El problema de minimización propuesto aparece en aquellos casos de ingeniería en los que hay que unir un conjunto de orígenes con un destino de tal modo que, por un lado, las distancias desde cada origen al destino sean mínimas (para mejorar la *calidad del servicio* o QoS, del inglés Quality of Service) y, por otro lado, los costes de construir la infraestructura sea también minimizada (lo cual reduce los *gastos de inversión* o IE, del inglés Investment Expenses). Esto puede suceder, por ejemplo, si se quiere construir una red de área local (LAN) con forma de árbol, de tal manera que todos los ordenadores de todas las oficinas y edificios de una empresa estén conectados al servidor central de la compañía. De forma similar, se puede considerar el mismo problema si queremos construir una red arborescente de autopistas para unir una serie de ciudades con la capital y queremos compensar dos objetivos: minimizar las distancias desde cada ciudad a la capital y minimizar el coste de construir el *árbol* de autopistas.

Como en los problemas MDV y MSV (ver la sección 4.2), generalizamos el nuevo problema propuesto para ampliar la capacidad de expresar situaciones más complejas del siguiente modo: consideramos que el coste de atravesar una arista del grafo depende del camino seguido hasta ese momento. Así, en el ejemplo de la LAN podemos definir aristas que sean

rápidas pero no muy fiables y que, por tanto, no sean adecuadas para la transmisión de datos provenientes de un nodo que normalmente ejecute aplicaciones en tiempo real pero,sin embargo, sí sean adecuadas para transmitir datos provenientes de un servidor de e-mail.

Probamos que el problema planteado es NP-completo (de hecho, generaliza un problema NP-completo conocido, el problema del *Árbol de Steiner mínimo*. Denotamos el problema planteado como el problema del *árbol QoS-IE* o QIT (del inglés, QoS-IE Tree problem). Definimos formalmente el problema y demostramos su NP-completitud. Para resolver este problema aplicamos el método RFD y una propuesta ACO, y comparamos los resultados. Observamos que cuando resolvemos el problema para valores de α intermedios, es decir, cuando queremos compensar los resultados obtenidos en QoS e IE, RFD mejora de forma clara los resultados obtenidos por ACO. De este modo queda patente la capacidad de RFD para conseguir soluciones *agrupadas y compensadas* ajustando tan sólo un parámetro del algoritmo en la forma de erosionar los caminos.

4.9. Testing Restorable Systems: Formal Definition and Heuristic Solution based on River Formation Dynamics

En [84] extendemos el trabajo previo desarrollado en [87]. En particular, se realizan experimentos más completos, se presenta un nuevo problema íntimamente relacionado con el MLS, el MRP, y se prueba la NP-completitud del MLS y del MRP. MRP se define como el problema de cubrir todas las configuraciones críticas cuando restaurar configuraciones previas no está permitido, es decir, cuando la única forma de volver a un estado que ya ha sido atravesado consiste en reiniciar el sistema y repetir el camino para ir desde el estado inicial hasta dicha configuración. Se asume que reiniciar el sistema también conlleva un coste.

Se extienden los algoritmos desarrollados en [87] para tratar explícitamente el caso en el que el conjunto de configuraciones críticas que deben alcanzarse al menos una vez no solo incluya nodos, sino también aristas. Empleando dichos algoritmos, se realizan numerosos experimentos sobre grafos de diferentes tamaños (50, 100 y 200 nodos) y formas (grafos dispersos y grafos densos). Se explican en detalle los algoritmos empleados en los experimentos: la adaptación del RFD para resolver los problemas MLS y MRP, y la implementación realizada de ramificación y poda.

Por último, aplicamos nuestra metodología de testing a un caso de estudio real. Queremos testear una aplicación de administración de una red social. Esta aplicación permite al administrador consultar y modificar los datos de los usuarios de la red. La funcionalidad del

sistema es definida empleando una máquina de estados finitos *extendida* (EFSM). Para testear esta aplicación generamos la FSM asociada a la EFSM que describe el sistema. Nuestro objetivo será recorrer al menos una vez todas las configuraciones críticas que definamos en el mínimo tiempo posible. Esta vez comprobamos con un caso práctico que guardar/restaurar configuraciones puede ahorrarnos tiempo a la hora de testear un sistema. Esto es especialmente significativo, pues el sistema considerado en el caso de estudio hace un alto uso de la memoria, por lo que guardar y restaurar son operaciones costosas.

Capítulo 5

Conclusiones y Trabajo Futuro

El principal objetivo de esta tesis ha sido desarrollar un nuevo método heurístico de optimización basado en la formación dinámica de los ríos. Para ello hemos realizado un estudio previo de distintos métodos heurísticos: la escalada, el enfriamiento simulado, los algoritmos genéticos, los algoritmos basados en nubes de partículas y los algoritmos basados en colonias de hormigas. De este estudio hemos extraído algunas de las características de estos métodos para sacar provecho de ellas y mejorar el comportamiento de nuestra propuesta. Concretamente, tomamos como principal punto de referencia los algoritmos basados en colonias de hormigas. Estos algoritmos han sido aplicados a diferentes problemas NP-completos con excelentes resultados, lo que nos permite comprobar la calidad de las soluciones de nuestro algoritmo.

Como se ha descrito en más detalle a lo largo de la tesis, el método propuesto funciona de la siguiente manera. Dado un grafo valorado, asociamos valores de altura a los nodos. Las gotas erosionan la tierra (es decir, reducen la altura de los nodos) o depositan sedimentos (es decir, aumentan la altura de los nodos) cuando se mueven de un nodo a otro. La probabilidad de que una gota seleccione una arista es proporcional a la pendiente de bajada en la arista, la cual depende de la diferencia de alturas entre ambos nodos y del coste de la arista. Al principio del algoritmo todos los nodos tienen la misma altura, a excepción del nodo destino que es un *hoyo* y representa el *mar*. Las gotas se depositan en el nodo origen y se esparcen por el grafo hasta que algunas de ellas llegan al nodo destino. Estos movimientos erosionan los nodos adyacentes creando nuevas pendientes y así se propaga el proceso de erosión. Después se insertarán nuevas gotas en el mismo nodo de origen para transformar los caminos y reforzar la erosión de aquellas rutas más prometedoras. Tras algunos pasos, se encontrarán buenos caminos desde el nodo origen al destino en forma de secuencias de aristas decrecientes en

altura.

Las principales ventajas de nuestro método consisten en que podemos evitar dos problemas intrínsecos de los métodos basados en colonias de hormigas. En primer lugar, en los métodos basados en colonias de hormigas, las hormigas pueden seguir rastros de feromonas de tal modo que, después de algunos movimientos, sea imposible no repetir un nodo del camino, es decir, han seguido un ciclo local. Destaquemos que esto no es posible en nuestra propuesta porque implicaría un ciclo *siempre decreciente*, lo cual es contradictorio. Si siempre vamos a un nodo de altura menor que el anterior no es posible formar un ciclo. Por otro lado, cuando las hormigas encuentran un nuevo camino más corto, es necesario que muchas hormigas seleccionen ese nuevo camino para *convencer* al resto de hormigas, que siguen un camino más antiguo muy reforzado de feromonas, a tomar este nuevo camino. Sin embargo, nuestra propuesta no presenta este problema. Al considerar las diferencias de alturas, cuando se halla un camino más corto, desde ese preciso momento sus aristas son preferibles (en media) a las aristas de otros caminos antiguos. Esto es debido a que la diferencia de alturas entre el nodo inicial y el final es la misma para todos los caminos, pero el coste será menor en el camino más corto. Por ello, la ratio *diferencia de alturas entre el origen y el destino/coste total del camino* será mayor en el camino más corto, lo que hará a dicho camino preferible para las gotas. Por el contrario, en los métodos basados en colonias de hormigas, las aristas del camino más corto no serán todavía preferibles puesto que la cantidad de feromonas presente en el nuevo camino será despreciable frente a los antiguos caminos aunque el coste del camino sea menor.

El método desarrollado ha sido aplicado en la resolución de diferentes problemas NP-completos. Uno de ellos es el problema del viajante de comercio, un problema clásico que ha sido estudiado en muchas investigaciones. Otros problemas NP-completos estudiados son el problema del árbol recubridor mínimo y el árbol de distancias mínimo en un grafo de costes variables. Estos nuevos problemas son generalizaciones de otros problemas conocidos de construcción de árboles en grafos. Además, hemos estudiado las versiones dinámicas de dichos problemas. Los resultados obtenidos han sido comparados con las soluciones halladas por un método basado en colonias de hormigas como ya dijimos anteriormente. Las conclusiones extraídas de estos experimentos son las siguientes: en general, a largo plazo nuestro método basado en la formación de los ríos obtiene mejores soluciones que los métodos basados en colonias de hormigas, mientras que las hormigas encuentran mejores soluciones más rápidamente.

El segundo objetivo de esta tesis ha consistido en aplicar el algoritmo desarrollado a métodos formales. Para ello realizamos un estudio de aplicaciones de distintos métodos de

computación evolutiva a los métodos formales. En este estudio observamos que los algoritmos evolutivos han sido recientemente introducidos para resolver problemas de testing de software y de model checking. La introducción de estos métodos a estos problemas es debida a que, a menudo, los métodos que realizan búsquedas óptimas no son aplicables en la práctica en los métodos formales, pues se encuentran con el problema de que el número de estados crece exponencialmente con el tamaño del sistema que se desea analizar. Es por ello que las técnicas exhaustivas son sustituidas por estrategias heurísticas para poder focalizar la búsqueda en configuraciones sospechosas de error o de importancia crítica.

En este área nos interesamos en la búsqueda de juegos de tests óptimos para recorrer, al menos una vez, algunos/todos los estados o transiciones de una especificación de un sistema definido por una máquina de estados finitos, donde el testeador del sistema puede guardar/restaurar cualquier configuración previa del sistema con un coste. La aplicación de nuestro método a resolver este problema NP-completo obtiene buenas soluciones en tiempos razonables. Además, mostramos con muchos experimentos que la opción de guardar/restaurar configuraciones es una buena opción cuando el coste de dicha operación es bajo en comparación con el coste que requiera típicamente alcanzar un nodo cualquiera del grafo desde otro nodo cualquiera del grafo, incluso en escenarios donde el coste de carga no es particularmente bajo. Para finalizar, observamos que nuestro método se adapta especialmente bien cuando se tiene que conseguir un compromiso entre: (a) minimizar el árbol de distancias mínimas y (b) minimizar el coste del árbol recubridor mínimo. Esto es debido a la siguiente propiedad de RFD. Si reducimos la erosión provocada por *grandes flujos*, entonces se reduce el incentivo de que las gotas traten de juntarse, lo que provoca que éstas tiendan a seguir su propio camino más corto. Para ello basta con establecer que n gotas que atraviesen juntas una arista provoquen la erosión de una única gota. Esto hace que el algoritmo tienda a hallar el árbol de distancias mínimas. En el otro extremo, si un flujo con n gotas realmente crea una erosión como lo harían n gotas individuales, entonces se fomenta la agrupación de gotas, lo que hace que el algoritmo tienda a hallar árboles de recubrimiento mínimo. Por último, si consideramos valores de erosión intermedios, entonces construimos árboles que encajan en los objetivos de los problemas (a) y (b).

En lo relativo al trabajo futuro, aún tenemos muchas tareas pendientes. Como RFD es un método joven, todavía se pueden introducir muchas mejoras al método básico para lograr superar la calidad de los resultados logrados hasta ahora. Estamos interesados en hacer que el número de gotas que recorren el grafo sea variable durante la ejecución, pudiendo de esta manera focalizar la exploración en algunas regiones; otra tarea pendiente es la de mejorar

el mecanismo de sedimentación que nos permite evitar caminos no deseados en los casos en que introducimos nodos barrera, que consistiría en depositar sedimentos no sólo en los nodos estándar, sino también en los nodos barrera; además queremos simular la *velocidad de las gotas*, ya que, intuitivamente, cuando una gota cae por una fuerte pendiente de bajada aumenta su velocidad, y esta energía cinética podría utilizarse para escalar pendientes de subida más tarde (es decir, es como si la velocidad fuese un tipo de *crédito* para poder subir pendientes).

Por otro lado, queremos realizar una implementación basada en colonias de hormigas para resolver el problema de la secuencia de carga mínima y comparar los resultados obtenidos por esta propuesta con los resultados que obtuvimos aplicando el método River Formation Dynamics, ya que la comparación de los resultados se realizó comparando nuestro método con una técnica exacta de ramificación y poda.

Siguiendo con el problema de buscar secuencias de testing que visiten al menos una vez una serie de configuraciones críticas, deseamos generalizar nuestro método para trabajar con máquinas de estados finitos extendidas. Como ya vimos al tratar los problemas MDV y MSV, RFD es capaz de tratar con *variables* sin necesidad de desplegar los estados de la máquina de estados finitos en todas las combinaciones de $(valor\ de\ la\ variable, estado)$. Esperamos poder abordar el problema de la mínima secuencia de carga en un contexto de máquinas de estados finitos extendidas tratando las variables como ya hicimos al tratar MDV y MSV.

Con respecto al método híbrido ACO-RFD, hemos probado su utilidad para resolver el problema del árbol de distancias mínimo en un grafo con costes variables, pero pensamos que este método podría ser especialmente prometedor para resolver otros problemas donde la combinación de las características de ambos métodos fuese una auténtica mejora. Por ello, queremos aplicar el método híbrido a resolver el problema del viajante de comercio, donde queremos obtener la rapidez a la hora de encontrar buenas soluciones del método ACO combinado con la mejora en la calidad de las soluciones halladas por RFD tras dejar pasar el suficiente tiempo.

Finalmente, nuestra última propuesta de trabajo futuro consiste en estudiar el comportamiento de RFD y de ACO para resolver otro problema NP-completo, el problema del árbol de Steiner mínimo, en el que el RFD podría obtener buenos resultados debido a la naturaleza del problema y la similitud a otros problemas resueltos.

Bibliografía

[1] Dining philosophers problem. http://en.wikipedia.org/wiki/Dining_philosophers_problem.

[2] Epistemowikia. http://campusvirtual.unex.es/cala/epistemowikia.

[3] Stable marriage problem. http://en.wikipedia.org/wiki/Stable_marriage_problem.

[4] UML. http://www.uml.org.

[5] E. Aarts and J. Korst. *Simulated Annealing and Boltzmann Machines: a stochastic approach to combinatorial optimization and neural computing*. John Wiley and Sons, 1989.

[6] E. Alba and F. Chicano. Finding safety errors with ACO. In *GECCO'07: 9th annual conference on Genetic and Evolutionary Computation Conference*, pages 1066–1073. ACM, 2007.

[7] E. Alba, F. Chicano, M. Ferreira, and J. Gomez-Pulido. Finding deadlocks in large concurrent java programs using genetic algorithms. In *GECCO'08: 10th annual conference on Genetic and Evolutionary Computation Conference*, pages 1735–1742. ACM, 2008.

[8] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

[9] M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Towards a tool environment for model-based testing with AsmL. In *FATES'03: Formal Approaches to Software Testing, LNCS 2931*, pages 252–266. Springer, 2003.

[10] D. Barstow. Artificial intelligence and software engineering. In *ICSE'87: 9th international conference on Software Engineering*, pages 200–211. IEEE Computer Society Press, 1987.

[11] B. Beizer. Software testing techniques. Technical report, Van Nostrand Reinhold Co., 1983.

[12] F. Belli and K.-E. Grosspietsch. Specification of fault-tolerant system issues by predicate/transition nets and regular expressions-approach and case study. *IEEE Transactions on Software Engineering*, 17(6):513–526, 1991.

[13] D.J. Berndt and A. Watkins. High volume software testing using genetic algorithms. In *HICSS'05: 38th Annual Hawaii International Conference on Systems Science*, pages 318–326. IEEE Computer Society, 2005.

[14] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.

[15] B.S. Bosik and M.Ü. Uyar. Finite state machine based formal methods in protocol conformance testing: from theory to implementation. *Computer Networks and ISDN Systems*, 22(1):7–33, 1991.

[16] K. Bouleimen and H. Lecocq. A new efficient simulated annealing algorithm for the resource-constrained project scheduling problem and its multiple mode version. *European Journal of Operational Research*, 149(2):268–281, 2003.

[17] L.C. Briand. On the many ways software engineering can benefit from knowledge engineering. In *SEKE'02: International Conference on Software Engineering and Knowledge Engineering*, pages 3–6. ACM, 2002.

[18] P.M.S. Bueno and M. Jino. Identification of potentially infeasible program paths by monitoring the search for test data. In *ASE'00: 15th IEEE international conference on Automated software engineering*, pages 209–218. IEEE Computer Society, 2000.

[19] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L. J. Hwang. Symbolic model checking: 10^20 states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[20] G. Di Caro and M. Dorigo. AntNet: Distributed stigmergic control for communication networks. *Artificial Intelligence Research*, 9:317–365, 1998.

[21] M. Clerc and J. Kennedy. The particle swarm - explosion, stability, and convergence in a multidimensional complex space. *IEEE Transactions on Evolutionary Computation*, 6(1):58–73, 2002.

[22] O. Cordón, I. Fernández de Viana, F. Herrera, and Ll. Moreno. A new ACO model integrating evolutionary computation concepts: The best-worst ant system. In *ANTS'00: From Ant Colonies to Artificial Ants: 2nd International Workshop on Ant Algorithms*, pages 22–29, 2000.

[23] O. Cordón, F. Herrera, and Ll. Moreno. Integración de conceptos de computación evolutiva en un nuevo modelo de colonia de hormigas. In *CAEPIA'99: VII Conferencia de la Asociación Española para la Inteligencia Artificial*, pages 98–105, 1999.

[24] T. Csondes, B. Kotnyek, and J.Z. Szabo. Application of heuristic methods for conformance test selection. *European Journal of Operational Research*, 142(1):203–218, 2002.

[25] L. Davis, editor. *Handbook of genetic algorithms*. Van Nostrand Reinhold New York, 1991.

[26] A. de la Encina, M. Hidalgo-Herrero, P. Rabanal, and F. Rubio. Applying evolutionary techniques to debug functional programs. In *IWANN'09: 10th International Work-Conference on Artificial Neural Networks, LNCS 5517*, pages 318–326. Springer, 2009.

[27] J. Deneubourg, S. Aron, S. Gross, and J. M. Pasteels. The self-organizing exploratory pattern of the argentine ant. *Insect Behavior*, 3(2):159–168, 1990.

[28] Q. Diller. *Z: An Introduction to Formal Methods*. John Wiley & Sons, Inc., 1994.

[29] K. Doerner and W.J. Gutjahr. Extracting test sequences from a markov software usage model by ACO. In *GECCO'03: 5th annual conference on Genetic and Evolutionary Computation Conference, LNCS 2724*, pages 2465–2476. Springer, 2003.

[30] M. Dorigo. *Ant Colony Optimization*. MIT Press, 2004.

[31] M. Dorigo and C. Blum. Ant colony optimization theory: a survey. *Theoretical Computer Science*, 344(2-3):243–278, 2005.

[32] M. Dorigo and G. Di Caro. Ant algorithms for discrete optimization. *Artificial Life*, 5(2):137–172, 1999.

[33] M. Dorigo and L. M. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

[34] M. Dorigo, V. Maniezzo, and A. Colorni. Ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, part B*, 26:29–41, 1996.

[35] I.K. El-Far and J.A. Whittaker. Model-based software testing. In *Encyclopedia on Software Engineering (edited by Marciniak)*. Wiley, 2001.

[36] A. Engel and M. Last. Modeling software testing costs and risks using fuzzy logic paradigm. *Journal of Systems and Software*, 80(6):817–835, 2007.

[37] M. Fleischer. Simulated annealing: past, present, and future. In *WSC'95: 27th Conference on Winter Simulation*, pages 155–161. IEEE Computer Society Press, 1995.

[38] C. Fritz. Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata. In *CIAA'03: 8th International Conference on Implementation and Application of Automata, LNCS 2759*, pages 35–48. Springer, 2003.

[39] P. Godefroid. Verisoft: A tool for the automatic analysis of concurrent reactive software. In *CAV'97: 9th International Conference on Computer Aided Verification, LNCS 1254*, pages 476–479. Springer, 1997.

[40] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In *TACAS'02: 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS 2280*, pages 266–280. Springer, 2002.

[41] D. E. Goldberg. *Genetic Algorithms (in Search, Optimization and Machine Learning)*. Addison Wesley, 1989.

[42] G. Gutin and A. P. Punnen. *The Traveling Salesman Problem and its Variations*. Kluwer, 2002.

[43] M. Harman and P. McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *ISSTA'07: International symposium on Software testing and analysis*, pages 73–83. ACM, 2007.

[44] R. L. Haupt and S. E. Haupt. *Practical Genetic Algorithms*. Wiley-Interscience, 2004.

[45] K. Havelund. Java pathfinder, a translator from java to promela. In *SPIN'99: 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking, LNCS 1680*, page 152. Springer, 1999.

[46] C. Heitmeyer, D. Mandrioli, and P. Milano. *Formal Methods for Real-Time Computing.* John Wiley & Sons, Inc., 1996.

[47] A. Hessel, K.G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. Testing real-time systems using UPPAAL. In *FORTEST'08: Formal Methods and Testing, LNCS 4949*, pages 77–117. Springer, 2008.

[48] M.G. Hinchey and S.A. Jarvis. *Concurrent Systems: Formal Development in CSP.* McGraw-Hill, Inc., 1995.

[49] M.G. Hinchey, C.A. Rouff, J.L. Rash, and W.F. Truszkowski. Requirements of an integrated formal method for intelligent swarms. In *FMICS'05: 10th international workshop on Formal Methods for Industrial Critical Systems*, pages 125–133. ACM, 2005.

[50] C.A.R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1):100–106, 1983.

[51] W.M.L. Holcombe. X-Machines as a basis for system specification. *Software Engineering*, 3(2):69–76, 1988.

[52] W.M.L. Holcombe. Mathematical models of cell biochemistry. *Molecular theories of cell life and death*, pages 250–263, 1991.

[53] W.M.L. Holcombe. Towards a formal description of intracellular biochemical organization. *Computers & mathematics with applications*, pages 107–115, 1991.

[54] G.J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

[55] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley Professional, 2003.

[56] R. Iosif. Symmetry reduction criteria for software model checking. In *SPIN'02: 9th International SPIN Workshop on Model Checking of Software, LNCS 2318*, pages 22–41. Springer, 2002.

[57] B.F. Jones, H.H. Sthamer, and D.E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.

[58] J. Kennedy and R. Eberhart. Particle swarm optimization. In *IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948. IEEE Computer Society Press, 1995.

[59] S. Khurshid. Testing an intentional naming scheme using genetic algorithms. In *TACAS'01: 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS 2031*, pages 358–372. Springer, 2001.

[60] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[61] H. Li and C.P. Lam. Optimization of state-based test suites for software systems: An evolutionary approach. *Int. J. Computer & Information Science*, 5(3):212–223, 2004.

[62] H. Li and C.P. Lam. Software test data generation using ant colony optimization. In *ICCI'04: International Conference on Computational Intelligence*, pages 1–4. International Computational Intelligence Society, 2004.

[63] A. Lluch-Lafuente, S. Edelkamp, and S. Leue. Partial order reduction in directed model checking. In *SPIN'02: 9th International SPIN Workshop on Model Checking of Software, LNCS 2318*, pages 112–127. Springer, 2002.

[64] N. López, M. Núñez, and I. Rodríguez. Assessing the expressivity of formal specification languages. In *AMAST'06: 11th International Conference on Algebraic Methodology and Software Technology, LNCS 4019*, pages 220–234. Springer, 2006.

[65] F. Luna, C. Blum, E. Alba, and A.J. Nebro. ACO vs EAs for solving a real-world frequency assignment problem in GSM networks. In *GECCO'07: 9th annual conference on Genetic and evolutionary computation*, pages 94–101. ACM, 2007.

[66] P. Mcminn and M. Holcombe. The state problem for evolutionary testing. In *GECCO'03: 5th annual conference on Genetic and Evolutionary Computation Conference, LNCS 2724*, pages 2488–2498. Springer, 2003.

[67] M.G. Merayo, M. Núñez, and I. Rodríguez. Formal testing from timed finite state machines. *Computer Networks*, 52(2):432–460, 2008.

[68] S. Merz. Model checking: a tutorial overview. In *MOVEP'00: Modeling and Verification of Parallel Processes, LNCS 2067*, pages 3–38. Springer, 2001.

[69] C.C. Michael, G. Mcgraw, and M.A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, 2001.

[70] J. Miller, M. Reformat, and H. Zhang. Automatic test data generation using genetic algorithm and program dependence graphs. *Information and Software Technology*, 48(7):586–605, 2006.

[71] J. Misra and K.M. Chandy. *Parallel Program Design: A Foundation*. Addison Wesley, 1988.

[72] M. Mitchell. *An Introduction to Genetic Algorithms*. The MIT Press, 1996.

[73] G.K. Palshikar. An introduction to model checking. Technical report, 2004. http://www.embedded.com.

[74] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 1982.

[75] R.P. Pargas, M.J. Harrold, and R.R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9:263–282, 1999.

[76] K.E. Parsopoulos and M.N. Vrahatis. Recent approaches to global optimization problems through particle swarm optimization. *Natural Computing*, 1(2-3):235–306, 2002.

[77] W. Pedrycz and J.F. Peters. *Computational Intelligence in Software Engineering*. World Scientific Publishing Co., Inc., 1998.

[78] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In *Current trends in concurrency, overviews and tutorials, LNCS 224*, pages 510–584. Springer, 1986.

[79] P. Rabanal and I. Rodríguez. Testing restorable systems by using RFD. In *IWANN'09: 10th International Work-Conference on Artificial Neural Networks, LNCS 5517*, pages 351–358. Springer, 2009.

[80] P. Rabanal, I. Rodríguez, and F. Rubio. Using river formation dynamics to design heuristic algorithms. In *UC'07: 6th international conference on Unconventional Computation, LNCS 4618*, pages 163–177. Springer, 2007.

[81] P. Rabanal, I. Rodríguez, and F. Rubio. Finding minimum spanning/distances trees by using river formation dynamics. In *ANTS'08: 6th international conference on Ant Colony Optimization and Swarm Intelligence, LNCS 5217*, pages 60–71. Springer, 2008.

[82] P. Rabanal, I. Rodríguez, and F. Rubio. Applying RFD to construct optimal quality-investment trees. Technical report, 2009. http://kimba.mat.ucm.es/-prabanal/research/jucs09.pdf, actualmente en proceso de revisión en la revista Journal of Universal Computer Science.

[83] P. Rabanal, I. Rodríguez, and F. Rubio. Applying river formation dynamics to solve NP-complete problems. In R. Chiong, editor, *Nature-Inspired Algorithms for Optimisation*, volume 193 of *Studies in Computational Intelligence*, pages 333–368. Springer, 2009.

[84] P. Rabanal, I. Rodríguez, and F. Rubio. Testing restorable systems: Formal definition and heuristic solution based on river formation dynamics. Technical report, 2010. http://kimba.mat.ucm.es/prabanal/research/stvr10.pdf, actualmente en proceso de revisión en la revista Software Testing, Verification and Reliability.

[85] P. Rabanal and I. Rodríguez. Hybridizing river formation dynamics and ant colony optimization. In *ECAL'09: 10th European Conference on Artificial Life*. Springer, in press.

[86] P. Rabanal, I. Rodríguez, and F. Rubio. Solving dynamic TSP by using river formation dynamics. In *ICNC'08: 4th International Conference on Natural Computation*, pages 246–250. IEEE Computer Society, 2008.

[87] P. Rabanal, I. Rodríguez, and F. Rubio. A formal approach to heuristically test restorable systems. In *ICTAC'09: 6th International Colloquium on Theoretical Aspects of Computing, LNCS 5684*, pages 292–306. Springer, 2009.

[88] J. Rech and K.D. Althoff. Artificial intelligence and software engineering: Status and future trends. *Künstliche Intelligenz*, 18(3):5–11, 2004.

[89] E. Rich and K. Knight. *Inteligencia artificial (Segunda edición)*. McGraw Hill Interamericana, 1994.

[90] I. Rodríguez. A general testability theory. In *CONCUR'09: 20th International Conference on Concurrency Theory, LNCS 5710*, pages 572–586. Springer, 2009.

[91] I. Rodríguez, M.G. Merayo, and M. Núñez. HOTL: Hypotheses and observations testing logic. *Journal of Logic and Algebraic Programming*, 74(2):57–93, 2008.

[92] S. Russell and P. Norvig. *Artificial Intelligence (A Modern Approach)*. Prentice Hall, 1995.

[93] K. M. Sim and W. H. Sun. Ant colony optimization for routing and load-balancing: survey and new directions. *IEEE Transactions on Systems, Man and Cybernetics, Part A*, 33(5):560–572, 2003.

[94] M.P. Singh, A.S. Rao, and M.P. Georgeff. Formal methods in DAI: logic-based representation and reasoning. In *Multiagent systems: a modern approach to distributed artificial intelligence*, pages 331–376. MIT Press, 1999.

[95] A.E.K. Sobel and M.R. Clarkson. Formal methods application: An empirical tale of software development. *IEEE Transactions on Software Engineering*, 28(3):308–320, 2002.

[96] T. Stützle and H. Hoos. Max-min ant system and local search for the traveling salesman problem. In *IEEE International Conference on Evolutionary Computation*, pages 309–314. IEEE Computer Society Press, 1997.

[97] T. Stützle and H. H. Hoos. Max-min ant system. *Future Generation Computer Systems*, 16(8):889–914, 2000.

[98] D.J.T. Sumpter, G.B. Blanchard, and D.S. Broomhead. Ants and agents: a process algebra approach to modelling ant colony behaviour. *Bulletin of Mathematical Biology*, 63:951–980, 2001.

[99] C. Tofts. Describing social insect behavior using process algebra. *Transactions on Social Computing Simulation*, pages 227–283, 1991.

[100] N. Tracey, J. Clark, J. McDermid, and K. Mander. A search-based automated test-data generation framework for safety-critical systems. In *Systems engineering for business process change: new directions*, pages 174–213. Springer, 2002.

[101] J. Tretmans. Testing concurrent systems: A formal approach. In *CONCUR'99: 10th International Conference on Concurrency Theory, LNCS 1664*, pages 46–65. Springer, 1999.

[102] J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7th European International Conference on Software Testing, Analysis and Review*, pages 8–12. EuroStar Conferences, 1999.

[103] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann, 2006.

[104] J.A. Whittaker and J.H. Poore. Markov analysis of software specifications. *ACM Transactions on Software Engineering and Methodology*, 2(1):93–106, 1993.

[105] J.A. Whittaker and M.G. Thomason. A Markov chain model for statistical software testing. *IEEE Transactions on Software Engineering*, 20(10):812–824, 1994.

[106] S. Wolfram. *Cellular Automata and Complexity.* Addison-Wesley, 1994.

[107] F. Zhang and T.Y. Cheung. Optimal transfer trees and distinguishing trees for testing observable nondeterministic finite-state machines. *IEEE Transactions on Software Engineering*, 29(1):1–14, 2003.

Apéndice A

Descripción de los Algoritmos Empleados

En este apéndice se describen en detalle los algoritmos empleados en las distintas implementaciones realizadas en cada uno de los artículos presentados en la tesis. También se describen los valores de los parámetros empleados en los experimentos para la resolución de cada problema.

A.1. Using River Formation Dynamics to Design Heuristic Algorithms

A.1.1. Algoritmo RFD-TSP

El algoritmo RFD empleado en [80] para resolver el TSP sigue el siguiente esquema:

```
initializeDrops()
initializeNodes()
while (not allDropsFollowTheSamePath())
       and (not otherEndingCondition())
   moveDrops()
   erodePaths()
   depositSediments()
   analyzePaths()
end while
```

Es importante recordar que para resolver el TSP es necesario que el nodo de origen sea clonado en dos nodos independientes con las mismas aristas: uno representará al nodo de origen y el otro hará las funciones de nodo destino. Sólo así RFD podrá crear soluciones en forma de pendientes de bajada entre un nodo de origen y un nodo destino. Una segunda característica especial respecto al esquema general para resolver este problema es que necesitamos encontrar un camino que visite todos los nodos del grafo. Por ello, las gotas necesitan memoria para recordar los nodos visitados. Como tercera característica especial respecto al esquema general, necesitamos utilizar nodos barrera (puede verse una descripción pormenorizada de estos nodos en [80]), los cuales son tratados de la misma manera que el resto de los nodos e intervienen en los procesos de inicialización, erosión y de sedimentación.

Veamos una descripción detallada de los pasos del algoritmo. Primero se inicializan las gotas en la fase `initializeDrops()`, donde se ponen todas las gotas en el nodo clonado que actúa como nodo inicial. En segundo lugar se inicializan los nodos en la fase `initializeNodes()`. En esta fase se realizan dos operaciones: en primer lugar se fija la altura del nodo destino a cero, mientras que la altura del resto de nodos (incluidos los nodos barrera) se establece a un mismo valor. En los experimentos se estableció una altura inicial de 10000.

El bucle `while` se ejecuta hasta que todas las gotas encuentran la misma solución (`allDropsFollowTheSamePath()`), es decir, hasta que todas las gotas siguen la misma secuencia de nodos desde el nodo de origen hasta el nodo destino o hasta que se satisface alguna otra condición de finalización (`otherEndingCondition()`). En particular, utilizamos esta condición para limitar el tiempo de ejecución o el número de iteraciones. Otra posible condición de finalización consiste en abortar la ejecución si la mejor solución encontrada no ha sido mejorada en las últimas n iteraciones. En nuestros experimentos, $n = 10000$.

El primer paso del cuerpo del bucle consiste en mover las gotas por el grafo (`moveDrops()`) de manera parcialmente aleatoria. En este paso se mueve cada gota hasta que encuentra una solución o hasta que queda estancada, ya sea porque todos los nodos vecinos ya han sido visitados, ya sea porque la gota está en un valle, no pueda ascender una pendiente (trepar) y tenga que depositar sus sedimentos. La siguiente regla de transición define la probabilidad de que una gota k en un nodo i elija a j como nodo destino:

$$P_k(i,j) = \begin{cases} \frac{gradient(i,j)}{\sum_{l \in V_k(i)} gradient(i,l)} & si \quad j \in V_k(i) \\ 0 & si \quad j \notin V_k(i) \end{cases}$$

donde $V_k(i)$ es el conjunto de nodos vecinos del nodo i que pueden ser visitados por la gota k y *gradient(i,j)* representa la pendiente existente entre los nodos i y j, y se define como sigue:

$$gradient(i,j) = \frac{altitude(i) - altitude(j)}{distance(i,j)}$$

donde *altitude(x)* es la altura del nodo x y *distance(i,j)* es la longitud de la arista que une el nodo i con el nodo j. Nótese que, al principio del algoritmo, la altura de todos los nodos (exceptuando el nodo destino) es la misma, por lo que la diferencia de alturas será 0. En este caso se establece una pequeña pendiente para que pueda moverse la gota. En particular, en este caso especial asumimos *altitude(i) - altitude(j) = 1.*

Dependiendo del valor de $P(k)$ se decide si existirán *gotas trepadoras*, es decir, gotas que puedan ascender pendientes. Este mecanismo funciona de la siguiente manera. Dada una gota k situada en el nodo i, se decide de manera aleatoria si k puede subir pendientes ascendientes de acuerdo a la siguiente probabilidad:

$$P(k) = \frac{1}{notClimbingFactor}$$

donde *notClimbingFactor* es una variable inicializada a 1 y que es incrementada ligeramente tras cada iteración (en la implementación del artículo se incrementa en 0,01 unidades). Cada N iteraciones (en el algoritmo $N = 100$), esta variable no será incrementada, sino que se disminuirá su valor en 0,5 unidades. En caso de que existan *gotas trepadoras* y que la diferencia de alturas sea menor que 0, establecemos la pendiente a $0{,}1/|gradient(i,j)|$, por lo que a mayor pendiente de subida, menor será la probabilidad de tomar ese camino. La regla de transición cuando existen gotas trepadoras queda definida como:

$$P_k(i,j) = \begin{cases} \frac{gradient(i,j)}{total} & \texttt{if} \quad j \in V_k(i) \\ \frac{\omega/|gradient(i,j)|}{total} & \texttt{if} \quad j \in U_k(i) \\ \frac{\delta}{total} & \texttt{if} \quad j \in F_k(i) \end{cases}$$

donde $\omega = 0{,}1$, $\delta = 1$ y $V_k(i)$, $U_k(i)$ y $F_k(i)$ son los conjuntos de nodos que son *vecinos* del nodo i que pueden ser visitados por la gota k y que están conectados mediante una pendiente de subida, bajada o llana, respectivamente.

En la siguiente fase (`erodePaths()`) se erosionan los caminos de acuerdo a los movimientos de las gotas de la fase anterior. La erosión se realiza una vez que cada gota ha encontrado una solución. En particular, si una gota se mueve en su camino desde el nodo i al j, entonces se erosiona el nodo i. La altura del nodo i se modifica de la siguiente manera:

$$altitude(i) = altitude(i) - erosion(i, j)$$

$$erosion(i, j) = \frac{paramErosion \cdot tamanioGota}{costeSolucion} \cdot gradient(i, j)$$

donde *paramErosion* es un parámetro (en los experimentos realizados toma el valor 0,1), *tamanioGota* es el tamaño de la gota y *costeSolucion* es el coste de la solución hallada por la gota. Se debe tener en cuenta que la altura del nodo final nunca se modifica y permanece igual a 0 a lo largo de toda la ejecución. La erosión producida se va acumulando, para que de este modo se cree la pendiente de bajada. Así, en un camino $i \to j \to k$ si el nodo i fue erosionado e unidades y a este nodo le sigue el nodo j, el nodo j será erosionado $e + erosion(j, k)$ y el nodo barrera existente entre j y k será erosionado $e + (erosion(j, k) \cdot 1, 5)$ unidades.

Una vez que ha finalizado el proceso de erosión, se incrementa levemente la altura de todos los nodos en la fase `depositSediments()`. El objetivo de este proceso es evitar que, después de muchas iteraciones, el proceso de erosión nos lleve a una situación donde todas las alturas estén cercanas a 0, lo que haría que las pendientes fuesen despreciables y arruinaría los caminos formados. En particular, la altura de un nodo i se aumenta de acuerdo a la siguiente expresión:

$$altitude(i) = altitude(i) + (paramSedim \cdot numGotas)$$

donde *paramSedim* toma el valor 1 y *numGotas* es el número de gotas utilizadas en el algoritmo (en estos experimentos se usaron 100 gotas). También permitimos que las gotas depositen sedimentos en un nodo. Esto ocurre cuando todos los movimientos posibles para una gota implican subir una pendiente y no lo logra de acuerdo a la probabilidad $P(k)$ asignada para ello. En este caso la gota queda bloqueada y deposita los sedimentos que transporta, incrementando la altura del nodo actual i de la siguiente manera:

$$altitude(i) = altitude(i) + paramBlockedDrop \cdot tamanio$$

donde $paramBlockedDrop = 0{,}1$ y *tamanio* es el tamaño de la gota que se ha quedado bloqueada.

Por último, en la fase `analyzePaths()` se selecciona la mejor solución encontrada por una de las gotas.

A.1.2. Algoritmo ACO-TSP

El algoritmo ACO utilizado en [80] para comparar los resultados obtenidos por RFD para el TSP, es un algoritmo inspirado en el algoritmo Ant System [34] que sigue el siguiente esquema:

```
initializeAnts()
initializePheromones()
while (not allAntsFollowTheSamePath())
       and (not otherEndingCondition())
   moveAnts()
   depositPheromones()
   evaporatePheromones()
   analyzePaths()
end while
```

En primer lugar, se inicializan las hormigas en `initializeAnts()`, donde se ponen todas las hormigas en el nodo inicial (el nodo que actuará como nido). En los experimentos realizados se emplearon 1000 hormigas, al observarse a lo largo de varias pruebas que era el valor óptimo. En `initializePheromones()` se establecen los valores iniciales de las feromonas presentes en las aristas (en los experimentos se estableció a 1000).

El bucle `while` se ejecuta hasta que todas las hormigas encuentran la misma solución (`allAntsFollowTheSamePath()`), es decir, hasta que todas las hormigas siguen el mismo camino desde el nodo de origen hasta el nodo destino, o hasta que se satisface alguna otra condición de finalización (`otherEndingCondition()`), que también puede utilizarse para limitar el tiempo de ejecución o el número de iteraciones.

La primera instrucción del cuerpo del bucle `moveAnts()` mueve cada una de las hormigas por el grafo hasta que encuentra una solución o hasta que queda bloqueada porque todos los nodos vecinos ya han sido visitados, situación en la cual ya no podrá encontrar una solución al problema porque tendría que volver a pasar por un nodo ya recorrido. La siguiente regla de transición define la probabilidad de que una hormiga k en un nodo i elija j como nodo destino:

$$P_k(i,j) = \begin{cases} \frac{pheromones(i,j)^{\alpha} \cdot distance(i,j)^{-\beta}}{\sum_{l \in V_k(i)} pheromones(i,l)^{\alpha} \cdot distance(i,l)^{-\beta}} & si \quad j \in V_k(i) \\ 0 & si \quad j \notin V_k(i) \end{cases}$$

donde $pheromones(i,j)$ representa la cantidad de feromona presente en la arista que une el nodo i con el nodo j, $distance(i,j)$ es la longitud de dicha arista y $V_k(i)$, es el conjunto de nodos vecinos del nodo i que pueden ser visitados por la hormiga k, es decir, los nodos vecinos que aún no han sido visitados. Por su parte, α y β son dos parámetros que ponderan el peso del rastro de feromona y de la distancia, respectivamente, en la decisión probabilística de la hormiga. En los experimentos, $\alpha = 1$ y $\beta = 5$.

En la siguiente fase (`depositPheromones()`) las hormigas que han encontrado una solución depositan feromonas en las aristas recorridas de manera proporcional a la solución encontrada:

$$pheromones(i,j) = pheromones(i,j) + paramDeposit/distance(i,j)$$

donde *paramDeposit* es la cantidad de feromonas que deposita una hormiga (en los experimentos realizados toma el valor 100).

En la fase `evaporatePheromones()` se evapora un porcentaje de las feromonas presentes de todas las aristas según la siguiente regla:

$$pheromones(i,j) = pheromones(i,j) \cdot (1 - paramEvaporate)$$

donde *paramEvaporate* es un parámetro que toma valores entre 0 y 1. En los experimentos realizados en el artículo $paramEvaporate = 0{,}5$.

Por último, en la fase `analyzePaths()` se selecciona la mejor solución encontrada por una de las hormigas.

A.2. Finding Minimum Spanning/Distances Trees by using River Formation Dynamics

A.2.1. Algoritmo RFD-MDV-MSV

Para resolver los problemas MDV y MSV empleando RFD se sigue el mismo esquema general que para resolver el TSP (ver sección A.1.1):

```
initializeDrops()
initializeNodes()
while (not endingCondition())
   moveDrops()
   erodePaths()
```

```
    depositSediments()
    constructSolution()
end while
```

En primer lugar se inicializan las gotas en la fase `initializeDrops()`, donde se ponen las gotas en todos los nodos excepto en el nodo destino. El número de gotas es un parámetro que puede establecer el usuario. En los experimentos se observó que este parámetro apenas tiene relevancia en los resultados obtenidos. Particularmente, se emplearon 10 gotas en cada nodo. En segundo lugar, se inicializan los nodos en la fase `initializeNodes()`. En esta fase se fija la altura del nodo destino a cero, mientras que la altura del resto de nodos se establece a un mismo valor. En los experimentos se utilizó una altura 1000.

En este algoritmo se manejan alturas diferentes para cada grupo de gotas según su nacimiento. Por ello, es necesario guardar las alturas de los nodos para cada uno de los nacimientos de las gotas. Las gotas sólo modificarán las alturas de los nodos asociadas a su nacimiento. Esta característica se debe tener en cuenta en el resto de las fases del algoritmo.

El bucle `while` se ejecuta hasta que se satisface alguna condición de finalización (`endingCondition()`). De nuevo, se puede limitar el tiempo de ejecución y/o el número de iteraciones.

El primer paso del cuerpo del bucle consiste en mover las gotas por el grafo (`moveDrops()`) de manera parcialmente aleatoria. En este paso se mueve cada gota una sola vez siguiendo la misma regla de transición aplicada para resolver el TSP (ver sección A.1.1). Recordemos que ahora las gotas tendrán en cuenta las alturas asociadas al nodo dónde nacieron y que el coste de la arista depende del camino previo seguido por la gota. El camino seguido previamente por cada gota determina el valor de una variable asociada a la gota. Dependiendo del valor de esa variable, cada arista tendrá un coste u otro. Una vez atravesada una arista, el valor de la variable es modificado empleando una función (de hecho, esta función forma parte de la definición del grafo de costes variables que estemos utilizando).

En la fase `erodePaths()` se erosionan los caminos de acuerdo a los movimientos de las gotas de la fase anterior. A diferencia de la versión implementada para resolver el TSP, la erosión se realiza justo después del movimiento de cada gota. En particular, si una gota se mueve en su camino desde el nodo i al j, entonces se erosiona el nodo i. La altura del nodo i se modifica de la siguiente manera:

$$altitude(i) = altitude(i) - erosion(i,j)$$

$$erosion(i,j) = \frac{paramErosion \cdot gradient(i,j)}{(numNodos - 1) \cdot numGotas}$$

donde *paramErosion* es un parámetro que en los experimentos toma valor 1, *numNodos* es el número de nodos del grafo y *numGotas* el número de gotas que *nacen* en cada nodo. Cabe destacar que en el proceso de erosión sólo se modificará la altura de los nodos *asociados* al nodo en el que *nació* la gota. En este paso se actúa de manera diferente dependiendo de que estemos resolviendo el MDV o el MSV. La regla de erosión anterior es la aplicada al resolver el MDV. En la resolución del MSV, se pretende que la erosión fomente la formación de caminos agrupados entre gotas de distintas procedencias para reducir el número de aristas a incluir en el árbol recubridor. Por contra, en el MDV se pretende que el árbol recubridor proporcione caminos mínimos desde cada nodo de origen al nodo destino. En el MSV la erosión producida será:

$$erosion(i,j) = \frac{paramErosion \cdot gradient(i,j)}{(numNodos - 1) \cdot numGotas} \cdot tamanio$$

donde *tamanio* es el tamaño de la gota que está produciendo la erosión. La suma de las erosiones producidas es almacenada en una variable.

En la fase `depositSediments()` se reparten los sedimentos erosionados por todas las gotas en la fase anterior a partes iguales entre todos los nodos.

Por último, en la fase `constructSolution()` se crea una solución al problema en función de los recorridos de las gotas. La formación de la solución difiere ligeramente entre el problema MDV y el MSV. Para crear una solución al MSV, se crea un árbol de la siguiente manera: se establece como raíz del árbol el nodo destino (el mar) y se van añadiendo aquellas aristas del grafo que unen a los nodos no incluidos en el árbol con éste. Se seleccionan en primer lugar aquellas aristas que fueron recorridas por un mayor número de gotas y que no crean un ciclo en el árbol. Este proceso se repite hasta que todos los nodos pertenecen al árbol. Para calcular el coste del árbol formado, como el coste de las aristas es variable, se toma como coste el *coste medio* de la arista. Como *coste medio* tomamos el valor de los costes de las gotas que atravesaron dicha arista y lo dividimos por el número de gotas que la atravesaron. Es decir, si la arista $i - j$ fue atravesada por 4 gotas con coste 3, por 7 gotas con coste 1 y también fue atravesada por 3 gotas con coste 5, el *coste medio* de la arista $i - j$ será $costeMedio(i,j) = (4 \cdot 3 + 7 \cdot 1 + 3 \cdot 5)/(4 + 7 + 3) = 34/14 \simeq 2{,}42$.

Nótese que, tanto en la resolución del MDV como en la resolución del MSV, las gotas tenderán a formar un árbol, es decir, una estructura sin ciclos. Si al comenzar en un determinado nodo y seguir las mayores pendientes de bajada completamos un ciclo, entonces alguna de dichas pendientes es de hecho de *subida*, lo que significa que el camino incluye un *valle*, por lo que el algoritmo todavía no ha formado los cauces hasta el mar y debe ejecutarse por

más tiempo. Por tanto, para formar la solución al MDV podemos formar el árbol partiendo de cada nodo (distinto del nodo destino) e ir añadiendo todas las aristas presentes en el camino de máxima pendiente hasta el mar. En el caso de que esta rama contenga algún ciclo no podremos hallar solución pues, como hemos dicho, el algoritmo todavía no habría creado los cauces apropiados hacia el mar. Al realizar este proceso desde cada nodo y unir todas las ramas generadas se formará una solución. En ocasiones, este árbol puede contener ciclos, habiéndose formado un subgrafo del grafo inicial. En este caso se aplicaría el método empleado para hallar una solución al MSV para eliminar dichos ciclos. Nótese que ahora se trabajaría sobre el subgrafo creado y no sobre el grafo inicial como se hacía en el caso del MSV.

A.2.2. Algoritmo ACO-MDV-MSV

El algoritmo ACO desarrollado para resolver estos problemas procuramos que siguiese el mismo esquema que el empleado en RFD:

```
initializeAnts()
initializePheromones()
while (not endingCondition())
   moveAnts()
   depositPheromones()
   evaporatePheromones()
   constructSolution()
end while
```

En primer lugar se inicializan las hormigas en la fase `initializeAnts()`, donde se ponen las hormigas en todos los nodos excepto en el nodo destino (que representa la fuente de comida). El número de hormigas es un parámetro que puede establecer el usuario. En los experimentos se observó que este parámetro apenas tiene relevancia en los resultados obtenidos. Se emplearon 10 hormigas en cada nodo. En segundo lugar se inicializan las feromonas iniciales en la fase `initializePheromones()`. En esta fase se establece el valor de feromonas inicial en cada arista. En los experimentos se utilizó un valor de 1000.

En este algoritmo se manejan feromonas diferentes para cada grupo de hormigas según su nacimiento. Por ello, es necesario guardar los rastros de feromonas de las aristas para cada uno de los nacimientos de las hormigas. Las hormigas modificarán los rastros de feromonas asociados a su nacimiento. Esta característica se debe tener en cuenta en el resto de las fases del algoritmo.

El bucle `while` se ejecuta hasta que se satisface alguna condición de finalización (`endingCondition()`). De nuevo, se puede limitar el tiempo de ejecución o el número de iteraciones.

El primer paso del cuerpo del bucle consiste en mover las hormigas por el grafo (`moveAnts()`) de manera parcialmente aleatoria. En este paso se mueve cada hormiga una sola vez siguiendo la misma regla de transición aplicada para resolver el TSP (ver sección A.1.2). Los parámetros α y β se establecieron a 4 y 2 respectivamente. Recordemos que ahora las hormigas tendrán en cuenta las feromonas asociadas al nodo dónde nacieron y que el coste de la arista depende del camino previo seguido por la hormiga. Este comportamiento es simulado de la misma manera empleada en el algoritmo RFD-MDV-MSV.

En la siguiente fase (`depositPheromones()`) las hormigas que han alcanzado el nodo destino depositan feromonas en las aristas recorridas de manera proporcional a la solución encontrada:

$$pheromones(i,j) = pheromones(i,j) + paramDeposit/distance(i,j)$$

donde *paramDeposit* denota la cantidad de feromonas que deposita una hormiga (en los experimentos realizados toma el valor 10). Se debe tener en cuenta que ahora el valor $distance(i,j)$ dependerá del valor de la variable manejada por cada hormiga en el recorrido que siguió para alcanzar el nodo destino.

En la fase `evaporatePheromones()` se evapora un porcentaje de las feromonas presentes en todas las aristas según la siguiente regla:

$$pheromones(i,j) = pheromones(i,j) \cdot (1 - paramEvaporate)$$

donde *paramEvaporate* es un parámetro que toma valores entre 0 y 1. En los experimentos realizados en el artículo $paramEvaporate = 0{,}01$. Este parámetro se establece a un valor muy bajo debido a que en el algoritmo se invoca a esta rutina tras cada movimiento de las hormigas.

Por último, en la fase `constructSolution()` se crea una solución al problema en función de los recorridos de las hormigas y las feromonas depositadas. La forma de crear una solución al problema es idéntica a la manera en el que se construía una solución en RFD. Para el caso del MSV se seleccionan en primer lugar aquellas aristas en las que hay mayor cantidad de feromonas (sumando el total de feromonas presente) y para el MDV se seleccionan aquellas aristas con mayor cantidad de feromonas, pero sólo teniendo en cuenta aquellas feromonas depositadas por las hormigas que fueron establecidas en la fase `initializeAnts()` en el nodo que comenzó la rama hasta el nodo destino.

A.3. Solving Dynamic TSP by using River Formation Dynamics

Para resolver el problema del TSP dinámico se extendieron los algoritmos empleados en [80] para poder trabajar con grafos dinámicos. De este modo se añadieron funciones para poder añadir y/o eliminar nodos y/o aristas.

Como ya se explica en [86], en los experimentos realizados en este artículo se siguen los siguientes pasos. Primero se calcula una solución de la instancia del problema usando los dos algoritmos (RFD y ACO) desarrollados en [80] y explicados en la sección A.1. A continuación se introduce alguno de los siguientes cambios en el grafo para comprobar la capacidad de reacción de cada grafo: (a) se elimina una arista común a las soluciones encontradas por ambos algoritmos; (b) se elimina un nodo del grafo; (c) se añade un nuevo nodo al grafo con sus correspondientes aristas. Una vez que se ha introducido un cambio, se vuelve a calcular una solución para la nueva instancia del problema. Cabe destacar que no se reinician los algoritmos tras introducir un cambio, sino que se mantiene el estado de cada unos de los métodos, es decir, se mantienen los niveles de feromonas de las aristas y las alturas de los nodos respectivamente.

A.4. Applying River Formation Dynamics to Solve NP-Complete Problems

Los algoritmos empleados para la obtención de los resultados mostrados en [83] son los desarrollados en [80, 81, 86]. Además se introducen resultados para las versiones dinámicas de los problemas MDV y MSV. En la implementación de estas nuevas versiones procedemos de igual modo que en la sección A.3, añadiendo a los algoritmos desarrollados en [81] la capacidad de trabajar con grafos dinámicos y así poder trabajar con la aparición y/o desaparición de nodos y/o aristas de la instancia del problema que se está resolviendo.

A.5. Hybridizing River Formation Dynamics and Ant Colony Optimization

El algoritmo híbrido HYB-ACO-RFD desarrollado en [85] emplea una nueva entidad para recorrer el grafo: la *hormiga-gota*. Esta entidad contiene todos los atributos de una hormiga así como todos los atributos de una gota. Este algoritmo sigue la misma estructura que los

algoritmos desarrollados en la sección A.2:

```
initializeAntsDrops()
initializeNodes()
initializePheromones()
while (not endingCondition())
    moveAntsDrops()
    depositPheromones()
    evaporatePheromones()
    erodePaths()
    depositSediments()
    constructSolution()
end while
```

A continuación describiremos las particularidades de este método. En la fase `initializeAntsDrops()` se inicializan las hormigas-gotas, donde se establecen éstas en todos los nodos excepto en el nodo destino. El número de gotas es un parámetro que puede establecer el usuario. En estos problemas este parámetro tiene poca influencia en los resultados obtenidos. Se emplearon 10 hormigas-gotas en cada nodo. En segundo lugar, se inicializan los nodos en la fase `initializeNodes()`. En esta fase se fija la altura del nodo destino a cero, mientras que la altura del resto de nodos se establece a un mismo valor. En los experimentos se utilizó altura 1000. En tercer lugar, se inicializan las feromonas iniciales en la fase `initializePheromones()`. En esta fase se establece el valor de feromonas inicial en cada arista. En los experimentos se utilizó un valor de 1000.

En este algoritmo se manejan rastros de feromonas y alturas diferentes para cada grupo de hormigas-gotas según su nacimiento. Por ello, de nuevo es necesario guardar las feromonas depositadas en las aristas y las alturas de los nodos para cada uno de los nacimientos de las hormigas-gotas. Las hormigas-gotas sólo modificarán los rastros de feromonas de las aristas y las alturas de los nodos asociadas a su nacimiento.

El resto de fases son idénticas a las descritas en la sección A.2 a excepción de algunas particularidades que comentamos a continuación. En la fase `moveAntsDrops()`, la probabilidad p_{ij} de que una hormiga-gota en el nodo i elija al nodo j como destino se calcula de la siguiente manera: primero se calcula la probabilidad p_{ij}^{ant} de que una hormiga elija la arista $i-j$, así como la probabilidad p_{ij}^{drop} de que una gota pudiera tomar dicha arista. Estas probabilidades se calculan del mismo modo que en la sección A.2, como si se tratase del método

ACO o RFD puro. Los parámetros α y β toman los valores 4 y 2 respectivamente. Entonces, la probabilidad p_{ij} de que la hormiga ubicada en i escoja ir a j queda definida como:

$$p_{ij} = p_{ij}^{ant} \cdot \omega_{ACO} + p_{ij}^{drop} \cdot \omega_{RFD}$$

donde ω_{ACO} y ω_{RFD} es el *peso relativo* que se da a cada uno de los métodos en la hibridación y $\omega_{ACO} + \omega_{RFD} = 1$. Estos pesos relativos no permanecen fijos a lo largo de la ejecución, sino que se actualizan en cada iteración. Así, ω_{ACO} comienza valiendo 1 y se va reduciendo hasta que toma el valor 0. Después, este valor se vuelve a incrementar poco a poco en cada iteración del bucle hasta que vuelve a tomar el valor 1.

En la fase `constructSolution()` se sigue el mismo método que el explicado en la sección A.2, donde a la hora de seleccionar la arista a introducir en el árbol se selecciona aquella arista por la que han pasado más hormigas-gotas.

Los algoritmos ACO y RFD con los que se comparan los resultados obtenidos por el método híbrido son los descritos en la sección A.2: RFD-MDV-MSV y ACO-MDV-MSV.

A.6. Testing Restorable Systems by using RFD

A.6.1. Algoritmo RFD-MLS

El algoritmo RFD desarrollado para la resolución del problema MLS sigue el mismo esquema que el mostrado en la sección A.2.1:

```
initializeDrops()
initializeNodes()
while (not endingCondition())
   moveDrops()
   erodePaths()
   depositSediments()
   joinDrops()
   constructSolution()
end while
```

A continuación describimos las diferencias de este algoritmo respecto al empleado en [81]. En la fase `initializeDrops()` se depositan las gotas iniciales sólo en aquellos nodos y/o aristas que queremos recorrer, es decir, en las configuraciones críticas, excepto el nodo destino

(en los nodos que no se consideren configuraciones críticas no *nacerán gotas*, es decir, no se hará llover en esos nodos). Como nodo destino o mar actuará el nodo inicial de la FSM. A continuación, se establece la altura de los nodos a 1000 en la fase `initializeNodes()`. En este algoritmo también se emplean nodos barrera al igual que el algoritmo empleado en [80] y también se inicializan en esta fase con altura 1000.

En la fase `moveDrops()`, cuando una gota alcanza el nodo destino, ésta se hace nacer en alguna de las configuraciones críticas elegida al azar. En esta versión sólo se maneja una altura para cada nodo, independientemente del nacimiento de la gota. A la hora de calcular el próximo movimiento de una gota, se penalizarán aquellos movimientos que conduzcan a un nodo donde ya haya sido movida una gota en ese turno, para de este modo penalizar que se realicen cargas (si se unen dos afluentes en la solución final, dicha solución deberá incluir una carga en ese punto). La penalización se realiza al calcular la pendiente para moverse del nodo i al j del siguiente modo:

$$gradient(i,j) = \frac{altitude(i) - altitude(j)}{distance(i,j) + loadCost}$$

donde *loadCost* es el coste de carga introducido como parámetro por el usuario.

Las fases `erodePaths()` y `depositSediments()` son equivalentes a las descritas en la sección A.2.1. Se añade una nueva fase: `joinDrops()`, en la que aquellas gotas que tras ser movidas se encuentran en el mismo nodo, se unen en una sola gota cuyo tamaño es la suma de los tamaños de las gotas unidas en el nodo.

Por último, en la fase `constructSolution()` se forma el árbol solución. Se comienza seleccionando la configuración crítica[1] que ha quedado con mayor altura y desde ésta se selecciona aquella arista cuya pendiente de bajada sea mayor. Los nodos del camino se van anotando como *visitados*. Así, si empezamos en el nodo i y j es el nodo al que lleva una mayor pendiente de bajada de entre los nodos vecinos de i, se añadirá la arista $i-j$ al árbol y desde j se seguirá el proceso extendiendo la rama hasta llegar al nodo destino o hasta llegar a un nodo que ya haya sido visitado. En el caso de que una rama llegue a un nodo k que haya sido visitado por otra rama previamente, se estará creando una carga (siempre y cuando dicho nodo no sea un nodo hoja en el árbol). En este caso, dependiendo del coste de carga y de la arista añadida, se estudiará si es mejor crear dicha carga o evitarla de alguna de las siguientes maneras: (i) uniendo la nueva rama con una nueva arista a un nodo que sea una hoja en el árbol (en vez de unir la rama con el nodo k) o (ii) uniendo el otro nodo

[1] El algoritmo RFD-MLS sólo está preparado para tratar configuraciones críticas que sean nodos (no está preparado para tratar configuraciones críticas que sean aristas).

implicado en la carga (el nodo que precede a k) a la hoja (es decir, el nodo inicial) de la nueva rama, quedando la nueva rama intercalada y unida a k.

A.6.2. Algoritmo B&B-MLS

El algoritmo de ramificación y poda desarrollado para resolver el problema MLS sigue el siguiente esquema:

```
for all (n = neighbourNotVisited(node))
   addToTheSolution(n)
   if isLeaf(node) then newLeaf(n,node)
   recursiveCall(n)
   recursiveCall(node)
end for
```

La primera llamada a esta función recursiva se realiza desde el nodo inicial (se establece como nodo inicial el estado inicial de la FSM) y finaliza cuando han sido visitadas todas las configuraciones críticas.[2] Además, son *podadas* aquellas ramas tales que:

$$costSolution + minCost < costBestSolution$$

donde $costSolution$ es el coste actual de la solución formada, $costBestSolution$ es el coste de la mejor solución hallada hasta el momento por el algoritmo y $minCost = notVisited \cdot cheapestEdge$ donde $notVisited$ es el número de configuraciones críticas que aún no han sido visitadas y $cheapestEdge$ es el coste de la arista más barata del grafo, por lo que $minCost$ representa el coste mínimo de añadir las configuraciones críticas aún no visitadas al árbol solución.

En la recursión se lleva control sobre los nodos y aristas visitadas, los nodos que son hojas en el árbol solución formado, el coste parcial de la solución y el número de cargas realizadas.

En el bucle `for all` se seleccionan aquellos nodos n vecinos de $node$ que no forman parte de la solución (`neighbourNotVisited(node)`). En `addToTheSolution(n)` se añade n a la solución, marcando el nodo n como visitado, la arista $n-node$ como visitada y añadiendo el coste de la arista $n - node$ al coste de la solución parcial.

[2] El algoritmo B&B-MLS, al igual que ocurría en el algoritmo RFD-MLS, sólo está preparado para tratar configuraciones críticas que sean nodos (no está preparado para tratar configuraciones críticas que sean aristas).

En la siguiente fase se comprueba si el nodo *node* es un nodo hoja en el árbol solución (`isLeaf(node)`). En caso afirmativo, *node* dejará de ser una hoja y n será una nueva hoja (`newLeaf(n,node)`) en el árbol construido.

Por último, se realizan las llamadas recursivas. En `recursiveCall(n)` se realiza la llamada recursiva desde el nuevo nodo introducido en el árbol n. En la llamada recursiva `recursiveCall(node)` se realiza de nuevo la llamada recursiva desde el nodo *node* (esta vez la arista $n - node$ ya ha sido visitada) creando una carga y sumando al coste de esta solución el coste establecido para las cargas $loadCost$.

A.7. A Formal Approach to Heuristically Test Restorable Systems

Los algoritmos empleados en [87] son los mismos que los descritos en la sección A.6.

A.8. Applying RFD to Construct Optimal Quality-Investment Trees

Los algoritmos empleados en [82] están inspirados en los utilizados en [81]. Tanto el método RFD como ACO siguen los mismos esquemas que los definidos en la sección A.2 para resolver el problema QIT.

A.8.1. Algoritmo RFD-QIT

En el algoritmo RFD utilizado para resolver el QIT se introducen los siguientes cambios respecto al esquema general mostrado en la sección A.2.1. En la fase `erodePaths()` la erosión producida será:

$$erosion(i,j) = \frac{paramErosion \cdot gradient(i,j)}{(numNodos - 1) \cdot numGotas} \cdot tamanio \cdot (1 - \alpha)$$

donde $0 \leq \alpha \leq 1$ es el parámetro de la expresión que se desea minimizar: $\alpha \cdot X + (1 - \alpha) \cdot Y$, siendo X la suma de los coste desde cada nodo de origen hasta el nodo destino a través del árbol solución e Y el coste del árbol, es decir, la suma de los costes de las aristas que forman el árbol. El resto de parámetros se establecen de la misma manera que en la sección A.2.1.

En la fase `constructSolution()`, para construir la solución se sigue el mismo método que para construir la solución en el problema MSV (ver sección A.2.1). Como se puede ver, la adaptación del RFD para resolver el QIT es muy sencilla.

A.8.2. Algoritmo ACO-QIT

En el algoritmo ACO utilizado para resolver el QIT se introducen los siguientes cambios respecto al esquema general mostrado en la sección A.2.2. En primer lugar, se ejecuta el algoritmo t_{QoS} unidades de tiempo, con la siguiente peculiaridad: en la fase `moveAnts()` se consideran diferentes tipos de feromonas según el *nacimiento* de las hormigas. A continuación, se sigue ejecutando el algoritmo t_{IE} unidades de tiempo, pero esta vez en la fase `moveAnts()` se considera que todas las feromonas son de un único tipo e independientes del nacimiento de las hormigas. Consideramos que $t_{QoS} = \alpha \cdot t_{total}$, $t_{QoS} + t_{IE} = t_{total}$ y t_{total} es el tiempo establecido por el usuario como parámetro. De este modo, se combinan las dos estrategias para conseguir buenas soluciones para optimizar QoS e IE.

En la fase `constructSolution()`, para construir la solución se sigue el mismo método que para construir la solución en el problema MSV (ver sección A.2.2).

A.9. Testing Restorable Systems: Formal Definition and Heuristic Solution based on River Formation Dynamics

Los algoritmos empleados en [84] se basan en los descritos en la sección A.7. Los algoritmos RFD-MLS y B&B-MLS han sido extendidos para poder tratar el uso de configuraciones críticas que sean aristas, dando lugar a los algoritmos RFD-MLS+ y B&B-MLS+ respectivamente, los cuales describiremos a continuación en las secciones A.9.1 y A.9.2.

A.9.1. Algoritmo RFD-MLS+

Este algoritmo sigue el mismo esquema que el mostrado en la sección A.6.1. Para el tratamiento de configuraciones críticas que son aristas sólo se ha modificado la fase *constructSolution*() como comentamos a continuación. En esta fase, se tratan todas las aristas críticas en primer lugar. De este modo, comenzamos a crear ramas de la solución comenzando en una *arista crítica* (elegida al azar) y terminando en el mar del mismo modo que en la sección A.6.1 se trataban los *nodos críticos*. También se utiliza el mismo método para evitar *cargas* no deseadas.

A.9.2. Algoritmo B&B-MLS+

El nuevo método de ramificación y poda para cubrir configuraciones críticas que sean tanto nodos como aristas sigue el siguiente esquema:

```
for all ((a → b) = edgeNotUsed())
    addToTheSolution(a → b)
    newLeaf(a,b)
    recursiveCall()
end for
```

Previamente a la recursión, se calcula una solución inicial trivial. Esta solución consiste en unir todas las configuraciones críticas directamente con el estado inicial, es decir, con el mar. Esta solución nos servirá para *podar* ramas no prometedoras desde el principio. A continuación se crea una solución vacía y se añade el nodo inicial, que hará el papel de raíz del árbol (se establece como nodo inicial el estado inicial de la FSM). La recursión finaliza cuando han sido visitadas todas las configuraciones críticas. Además, son *podadas* aquellas ramas tales que:

$$costSolution + minCost < costBestSolution$$

donde $costSolution$ es el coste actual de la solución formada, $costBestSolution$ es el coste de la mejor solución hallada hasta el momento por el algoritmo y $minCost = notVisited \cdot cheapestEdge$ donde $notVisited$ es el número de configuraciones críticas que aún no han sido visitadas y $cheapestEdge$ es el coste de la arista más barata del grafo, por lo que $minCost$ representa el coste mínimo de añadir las configuraciones críticas aún no visitadas al árbol solución.

En la recursión se lleva control sobre los nodos y aristas visitadas, los nodos que son hojas en el árbol solución formado, el coste parcial de la solución y el número de cargas realizadas.

En el bucle `for all` se seleccionan aquellas aristas $a \rightarrow b$ que no forman parte de la solución y tales que el nodo b ya esté incluido en la solución (`edgeNotUsed()`). En `addToTheSolution(`$a \rightarrow b$`)` se añade la arista $a \rightarrow b$ a la solución, marcando el nodo a como visitado, la arista $a \rightarrow b$ como visitada y añadiendo el coste de la arista $a \rightarrow b$ al coste de la solución parcial.

En la siguiente fase (`newLeaf(a,b)`) el nodo b dejará de ser una hoja (es posible que no lo fuese ya) y a se marcará como nodo hoja en el árbol construido. En caso de que b no fuese un nodo hoja, se creará una carga y se sumará el coste de carga al coste de la solución parcial.

Por último, se realiza la llamada recursiva con la nueva solución parcial (`recursive-Call()`).

A.9.3. Algoritmo RFD-MRP

Para adaptar RFD a MRP se mantienen las mismas ideas que en la sección A.9.1 para aplicar RFD a MLS. La principal modificación respecto al algoritmo RFD-MLS+ consiste en modificar el incentivo negativo para formar puntos de carga (en este caso *puntos de reset*). Dicha penalización se realiza al calcular la pendiente para moverse del nodo i al j (cuando al nodo j ya se ha movido alguna gota en el mismo turno) del siguiente modo:

$$gradient(i,j) = \frac{altitude(i) - altitude(j)}{distance(i,j) + resetCost + distance(0,j)}$$

donde *resetCost* es el coste de reiniciar la aplicación volviendo al estado inicial. Este valor es un parámetro introducido por el usuario.

Lo que se pretende es penalizar puntos de reset *lejanos* del nodo inicial, ya que, generalmente, recuperar esos estados será costoso. Se desea formar árboles solución tales que todas sus ramas se unan en la raíz, es decir, en el nodo inicial.

Además, la fase *constructSolution*() se ha modificado ligeramente. Cuando se ha de decidir si crear un punto de reset o tratar de evitarlo, el nodo n donde se crea la carga se sustituye por el nodo 0, el nodo inicial. De este modo, si se tuviese que crear una carga para volver hasta n, se utilizará el camino más corto entre el nodo inicial y n, en vez de la rama que se había creado anteriormente hasta n.